conference

proceedings

Proceedings of MobiSys 2003: The First International Conference on Mobile Systems, Applications, and Services

San Francisco, CA, May 5–8, 2003

MobiSys 2003

The First International Conference on Mobile Systems, Applications, and Services

San Francisco, CA, USA
May 5–8, 2003

Jointly sponsored by
ACM SIGMOBILE and
The USENIX Association,
in cooperation with **ACM SIGOPS**

acm

sigmobile

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $24 for members and $30 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

ISBN 1-931971-09-9

Printed in the United States of America on 50% recycled paper, 10–15% post-consumer waste.

Proceedings of

MobiSys 2003

The First International Conference on Mobile Systems, Applications, and Services

Jointly sponsored by ACM SIGMOBILE and
The USENIX Association,
in cooperation with ACM SIGOPS

May 5–8, 2003
San Francisco, CA, USA

Conference Organizers

General Chair
Dan Siewiorek, *Carnegie Mellon University*

Program Committee
Co-chair: Mary Baker, *Stanford University*
Co-chair: Robert T. Morris, *Massachusetts Institute of Technology*
Gaetano Borriello, *University of Washington and Intel Research Seattle*
Claude Castelluccia, *INRIA Rhone-Alpes*
Hao Chu, *DoCoMo USA Labs*
David Culler, *University of California, Berkeley, and Intel Research Berkeley*
Nigel Davies, *Lancaster University and University of Arizona*
Maria Ebling, *IBM T. J. Watson Research Center*
Peter Honeyman, *CITI, University of Michigan*
Cristina Hristea, *Stanford University*
Tim Kindberg, *Hewlett-Packard Labs, Palo Alto*
David Kotz, *Dartmouth College*
Kevin Lai, *University of California, Berkeley*
Adrian Perrig, *Carnegie Mellon University*
Doug Terry, *Microsoft Research, Silicon Valley*
Marvin Theimer, *Microsoft Research*
Henry Tirri, *University of Helsinki*
Amin Vahdat, *Duke University*
Deborah A. Wallach, *Google*

Tutorial Chair
Songwu Lu, *University of California, Los Angeles*

Research Demo Chair
Tom Martin, *Virginia Tech*

Steering Committee Chair
Victor Bahl, *Microsoft Research*

The USENIX Association Staff

External Reviewers

Imad Aad	Carla Ellis	Tayed Lemlouma	Vincent Roca
Joel Bartlett	Thierry Ernst	Bachun Li	Mema Roussopoulos
Mark Billinghurst	Alberto Escudero-Pascual	Jinyang Li	Sumit Roy
Marion Blount	Ken Fishkin	Athina Markopoulou	Mehran Sahami
Prasad Boddupalli	Adrian Friday	Robert N. Mayo	Dan Simon
Dan Boneh	Sanjay Ghemawat	Malena Mesarina	Dawn Song
Philippe Bonnet	TJ Giuli	Greg Minshall	Oliver Storz
Lawrence S. Brakmo	Vipul Gupta	David Mizell	Chandu Thekkath
Jack Brassi	Urs Hengartner	Gabriel Montenegro	Thierry Turletti
Torsten Braun	Jamey Hicks	Jeff Morgan	Amit Vyas
Claude Chaudet	David Hole	Brian R. Murphy	Susie Wee
Detlef Clawin	Wilson Hsieh	Pekka Nikander	Ted Wobber
Norm Cohen	Yih-Chun Hu	Ken Ohta	Dongyan Xu
Sunny Consolvo	Michael Isard	Shankar Ponnekanti	Lidong Zhou
Douglas De Couto	Brad Karp	Kimmo Raatikainen	
John S. Davis II	Hahnsang Kim	Parthasarathy Ranganathan	
David desJardins	Dejan Kostic	Venugopalan Ramasubramanian	
Andrezj Duda	Sacha Krakowiak	Patrick Reynolds	
Keith Edwards	Lea Kutvonen	Christophe Rippert	
Christos Efstratiou	David Lee		
Karim El-Malki	Sung Ju Lee		

MobiSys 2003: The First International Conference on Mobile Systems, Applications, and Services

May 5–8, 2003
San Francisco, CA, USA

Wednesday, May 7, 2003

Index of Authors

Message from the Program Chairs

We are delighted to welcome you to The First International Conference on Mobile Systems, Applications, and Services—MobiSys 2003. We are very gratified to see the final fruit of so much effort from so many people interested in bringing you this exciting new conference.

Given that this is the first year for MobiSys, we were surprised, honored, and terrified to receive such an enthusiastic response from researchers: 153 papers! The task of choosing the 23 best from among these took a tremendous amount of time and dedication from our program committee, to whom we extend our sincerest thank you. Through a long review and email discussion process, with over 475 written reviews produced, the committee narrowed down our proposed selections to a little over 50 papers before we met for a full day in Boston, Massachusetts, in early December 2002 to select the final papers for the proceedings. At the end of this meeting, the committee's work was not done: all accepted papers were assigned program committee shepherds to help the authors get the most out of the review comments and suggestions and to ensure the highest quality proceedings. Finally, program committee members also chair the technical sessions and select the winner of the Best Paper Award. We were very impressed with the efficiency, reliability, graciousness, and insight of our program committee members. What a pleasure it has been to work with them!

In addition we were lucky to receive so much valuable help and benefit of experience from the officers of the ACM Special Interest Group on Mobility of Systems, Users, Data and Computing (Sigmobile), and from Victor Bahl and David Johnson in particular. Thank you! This conference is a collaboration between the ACM and USENIX, and our Steering Committee Chair, Victor Bahl, spent many hours making this possible.

We also thank Tom Martin and Songwu Lu for helping to put together an excellent research demo session and tutorial program, respectively. We thank Bob Brodersen of the Berkeley Wireless Research Center and the University of California at Berkeley for giving a stimulating keynote address. We thank Dan Siewiorek, our general chair, for his many helpful answers to many questions, and our USENIX Board Liaisons, Peter Honeyman and Mike Jones, for all their excellent advice and support. The efforts of these people have contributed greatly to an overall exciting and diverse program for MobiSys.

We extend a special thank you to all of the USENIX staff for their cheerful and immediate response to our many naive questions and difficult requests. Ellie Young, Jane-Ellen Long, and Jennifer Radtke in particular made our work so much easier. They are seemingly tireless!

Finally, we thank all of the authors who took a risk sending their material to this new conference, regardless of whether we were able in the end to include it in the proceedings. The originality and practicality of so many of the papers received show that this is indeed a thriving, growing field with more wonderful work to come in the years ahead.

We hope you find MobiSys 2003 in San Francisco, CA, to be as interesting and enjoyable as we do.

Mary Baker, Stanford University
Robert Morris, Massachusetts Institute of Technology
MobiSys 2003 Co-Chairs

Message from the General Chair

The opportunity to inaugurate a major new conference is a rare event. The confluence of a number of technologies makes the time ripe to found a conference dedicated to building mobile systems. Mobile hardware platforms, light-weight operating systems, wireless networking, and new application paradigms have enabled the creation of systems that were written about in science fiction only a decade ago.

Above all, MobiSys is a conference about systems, not just their theory but their implementation. Among the top researchers in the field worldwide are presenting their latest results. There are outstanding tutorials in cutting-edge topics conducted by researchers who helped formulate the subject areas and have made seminal contributions.

The response from the system's community has been outstanding, as is detailed by Mary Baker and Robert Morris in their Message from the Program Chairs. Mary and Robert have done a magnificent job in mobilizing the community to review the papers and in setting a standard for high quality that will be the goal of future MobiSys program committees. Although MobiSys 2003 is a new event, they have produced a program that has the intensity and quality of a mature conference.

A sincere note of thanks to Victor Bahl, Steering Committee Chair, who envisioned MobiSys and orchestrated its birth. Also a special thank you to our sponsoring organizations, ACM and USENIX.

All of the organizers hope you enjoy the first edition of MobiSys 2003, the International Conference on Mobile Systems, Applications, and Services.

Dan Siewiorek, Carnegie Mellon University
MobiSys 2003 General Chair

Single Reflection Spatial Voting: A Novel Method for Discovering Reflective Surfaces Using Indoor Positioning Systems

R. K. Harle
Laboratory for Communication Engineering, University of Cambridge, UK
rkh23@cam.ac.uk

A. Ward
Ubiquitous Systems Limited
andy.ward@ubiquitous-systems.com

A. Hopper
Laboratory for Communication Engineering, University of Cambridge, UK
hopper@eng.cam.ac.uk

Abstract

We present a novel method of using reflected pulses in indoor ultrasonic positioning systems to infer the details of reflective objects in the environment. The method is termed Single Reflection Spatial Voting (SRSV), and we perceive its major use to be in the field of pervasive computing, where automated object and surface discovery is emerging as an important feature.

We demonstrate use of the method in the case of searching for vertical walls using an existing ultrasonic position sensor system (the Bat system). We find that valuable information can be extracted from reflection data using SRSV, and are able to construct a model of the room using a simple algorithm. We conclude that this method can be used to extract base data upon which to build hypotheses about the environment, given further sensor analysis.

We also briefly address the potential uses of SRSV in Ultra-Wideband positioning, autonomous navigation, and map building.

1 Introduction

Much of the current research in location systems concentrates on creating an accurate and reliable indoor positioning system, the perceived use of which is as a major component of a pervasive, context-aware computing system. Indoor positioning systems typically rely on the propagation of a physical wave phenomenon, such as ultrasonic or radio waves. The prevalence of reflecting surfaces in indoor environments drastically reduces the accuracy and reliability of positions returned from such systems. We present a method to turn this apparent weakness into a potential strength. We use reflected signals to help automate the process of modelling the environment within which the sensor network is operating.

Our experiences with a context-aware system [1] tell us that useful and reliable applications stem from having both an accurate position for an object or person, *and* from knowing details about the environment within which they are positioned. Presently, such environmental information is painstakingly entered by hand (for example, wall vertices, table positions, workstation locations) to allow a meaningful virtual model of the world to be formed (see Figure 1).

Once a useful model is established, it must also cope with the inherent dynamic nature of human environments. The applicability of a model diminishes rapidly as it loses synchronisation with the world. To remain useful, context-aware systems must be able to adapt to changing environments. Our experiences suggest that such adaptation can only stem from the coordination of sensor systems and processing methods. In this paper, we present a powerful method based on the principle that important environment information is superimposed on the positioning signals, in particular the reflections.

Given details of reflecting surfaces, we can de-

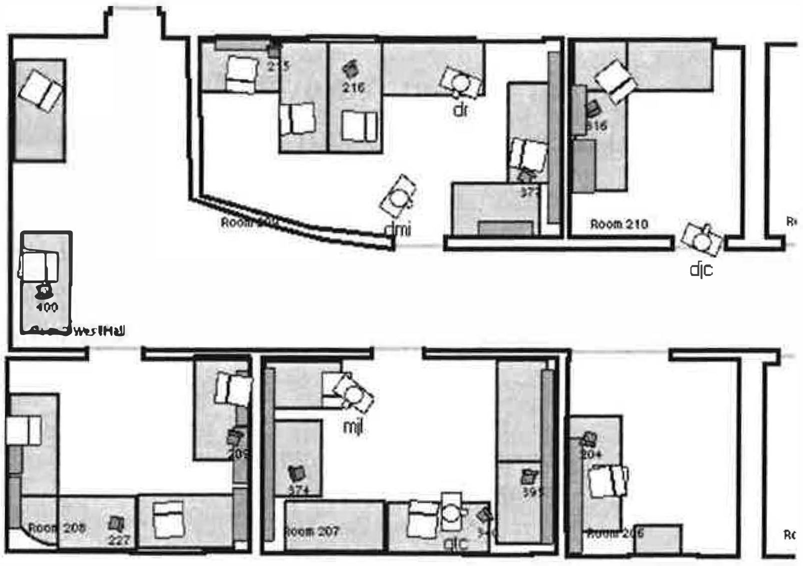


Figure 1: A screen shot from a real-time environment monitoring application in daily use.

velop better models of the world, allow for dynamic reconfiguration of the environment, and potentially provide feedback into a positioning system to improve accuracy. We stress that, whilst we present much of the paper in the context of an ultrasonic positioning system (the Bat system), the general method has potential uses outside of ultrasonic systems.

The essential requirement of the method is that the positioning medium interacts with objects within the environment, giving rise to reflected signals. Standard media used in positioning systems to date (ultrasound [16, 12, 14], radio [2, 3]) all suffer from environmental reflections, and this has traditionally been a large source of error. In addition, technologies expected to offer ubiquitous positioning in the future, such as Ultra Wideband radio, are susceptible to reflections.

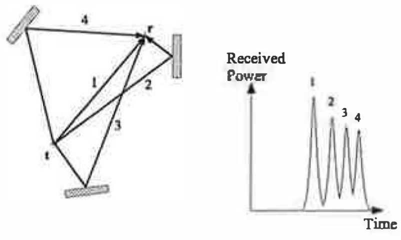
2 The Bat Ultrasonic Location System

The Bat system is a real-time positioning system for people and objects in an indoor environment [16]. The system has previously been implemented as a major component of a context-aware, pervasive computing system [1]. One installation of the Bat system covers the upper floor of the Laboratory for Communication Engineering, encompassing an area of approximately 550m^2, and provides a true context-aware computing environment in daily use.

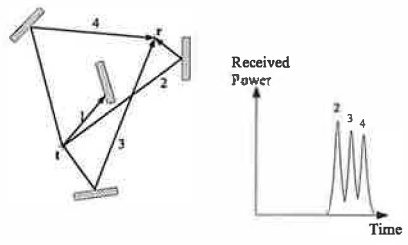
As described in [1], the Bat system is based around wireless active tags ("Bats") worn by users or attached to objects. A radio signal from a central controller triggers each Bat in turn, at a known time. When triggered, each Bat emits an ultrasonic pulse, which is received by a matrix of receivers placed at accurately known locations in the ceiling. By measuring the time delays between emission and reception of pulses, we gain estimates of the distance travelled by the pulse to each receiver within range. We then use a multilateration algorithm to convert the distance measurements to a location for the Bat in three-dimensional space [10].

2.1 Identifying Reflections Through Multilateration Algorithms

The Bat system operates in indoor office-like environments, which contain many objects and surfaces that specularly reflect ultrasound (such as computer monitors and walls). The pulse emitted by a Bat may travel directly to a receiver (a



(a) Three obstructions give rise to a direct-path signal peak (1) and three multipathed peaks (2,3,4) in the power profile of the received signal



(b) A further obstruction hinders the direct-path signal

Figure 2: Illustration of multipathed signal propagation between a receiver, r, and a transmitter, t

direct-path signal), but may also arrive after a reflection from one or more objects or surfaces in the environment (a multipathed signal). Multipathed signals do not provide information about the true (direct-path) distance of the Bat from the receiver, and hence it is important to identify and eliminate them from further processing.

In situations where a direct-path signal reaches a receiver, the Bat system can be sure that any subsequent pulse arrivals correspond to multipathed signals, because any reflected paths must necessarily be longer than the direct-path (Figure 2(a)). However, it is not always the case that the direct-path signal reaches the receiver, and in these situations the first signal to arrive at the receiver will, in fact, be a multipathed signal (Figure 2(b)). The positioning algorithm used in the Bat system is therefore designed to identify and reject multipathed signals, and we present here a simplified overview of that algorithm and the heuristics it uses (further details are available in [16]).

Following the emission of a pulse by a Bat, and its subsequent reception, we have a series of time-of-flight measurements. We convert these time measurements to distances using the well-documented temperature variation of the speed of sound [6]. We wish to derive four quantities from this information; the components of the Bat location (x,y,z), and an estimate of the positioning standard error. To do so, then, we require a minimum of four (non-multipathed) readings.

The process of identifying and excluding multipathed signals begins by forming a nonlinear model [13] of all the data, providing an estimate of the four quantities we are interested in. The error estimate is used to evaluate the model fit. If it exceeds the known accuracy of the system (approximately 3cm), we evaluate the residuals for each distance measure and order them numerically. Traditional nonlinear model algorithms order the magnitude of these residuals and discard the data associated with the largest and reprocess, thereby removing the outliers and improving the result. We use a different approach to capture the physical principle that no signal may arrive at a receiver early (corresponding to movement faster than the speed of sound), but only late (corresponding to a multipathed measure).

If we define $d_{measured}$ as the measured distance from a Bat to a particular receiver, d_{model} as the distance calculated by the model for the same quantity, then we can define the residual, e, as being directly proportional to

$$e \propto (d_{measured} - d_{model}) \qquad (1)$$

If we assume that the model has converged spatially close to the correct position, this quantity is negative when the model requires the signal to exceed the speed of sound, and positive when the signal appears to be multipathed (relative to the model predictions). We can adopt this as a general heuristic, since it fails only in the unlikely case of the majority of received signals being multipathed due to a specular object, causing the model to converge on a reflection image.

Since the speed of sound is fixed, but multipathed signals are likely, we maintain the sign of the residual, and discard the largest positive residual as multipathed. We repeat the process of modelling and discarding the measurement associated with the largest positive residual until we reach the required accuracy level, or until there are less than four measures remaining (a failure). Consequently, when we return a position, we have classified the initial distance measures into two groups; consistent with the position (assumed non-multipathed), and inconsistent (assumed multipathed). The Bat system receivers are also capable of recording the time-of-arrival of any *second* pulse they receive (which corresponds, as described earlier, to a multipathed signal). With each position, then, we can identify multipathed signals by:

1. Rejection from the positioning algorithm as described above.

2. Reception of a second pulse at a receiver that returned a first pulse consistent with the position calculated.

Whereas the Bat system rejects this reflection information as being of no use in computing location information, we now turn our attention to a method of using this data to derive information about the environment.

3 Single Reflection Spatial Voting (SRSV)

Barshan has described a spatial voting scheme for determining *two dimensional* surface profiles using ultrasonic rangefinders mounted on a mobile robot [5, 4]. The method associates a circular arc centred on the known position of the ultrasonic transceiver for each range measure, with a radius equal to half the distance covered by the ultrasonic pulse. By collecting a series of results from sufficiently different transceiver positions (generated by a number of rangefinders on the robot and/or by moving the robot platform), we expect to observe the highest density of arc intersections along the locus of the surface, as illustrated in Figure 3. Spatial voting splits the two dimensional plane into a regular grid of cells. With each cell, we associate a number representing the number of arcs which intersect its bounds. We can then extract a representation of the surface, quantised to the cell size, by searching the grid for high densities of intersections.

We present here a novel and significant extension to Barshan's method which uses a *three dimensional* Single Reflection Spatial Voting (SRSV) grid. This approach differs from that of

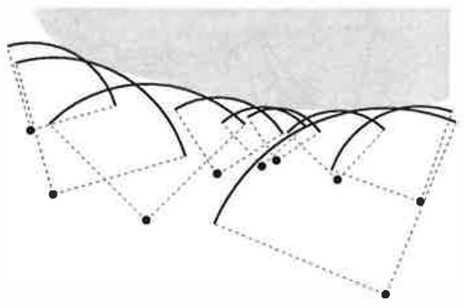


Figure 3: An illustration of the two dimensional surface extraction based on rangefinders (shown as black circles). The grey area represents the surface of interest, and the arc of each circle segment corresponds to the range reading from a specific rangefinder.

Barshan, since we use a physically distinct ultrasonic transmitter and receiver, an infrastructure that is in place and static, and a series of only multipathed measurements. SRSV is particularly suited to the pervasive computing environment because it allows incremental updating as more reflections are captured, but does not require that the details for each reflection be individually stored, thereby reducing storage requirements. We make the assumption that multipathed pulses have reflected only once; we have demonstrated that the method is sufficiently robust that multiple reflections do not render the method invalid in practice. In three dimensions, with a separate receiver and transmitter, the two dimensional circular arcs described by Barshan become prolate spheroids with foci at each of the receiver and transmitter locations, and the SRSV grid segments the volume into regular cubes ("cells"). Analysis of the density of spheroid intersections in each cell provides data about the environment, such as the locations of walls and specular objects.

3.1 The Prolate Spheroid of Reflection

Consider two points in three dimensional space representing a transmitter, $\mathbf{t}$, and a receiver, $\mathbf{r}$, and a signal that propagates between them via a single specular reflection from a point on a surface, $\mathbf{P}$. If the distance travelled by the signal is known (through time-of-flight measurement for example), the allowed locus of the reflection point is a prolate spheroid with major axis b and

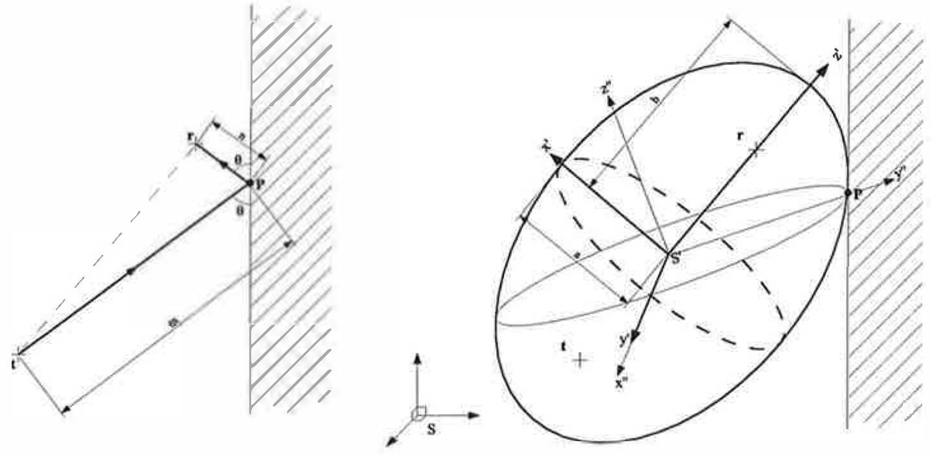


Figure 4: The reflection geometry for a single reflection from a point $\mathbf{P}$ between a transmitter at $\mathbf{t}$ and a receiver at $\mathbf{r}$, with a multipathed length of $l = m + n$. The thick outline (right) shows the associated prolate spheroid, and the thin outline shows the elliptical locus of $\mathbf{P}$ if we assert that it is on a *vertical* wall.

minor axis a (the shape formed by rotation of a two dimensional ellipse about its major axis), as shown in thick outline in Figure 4. Such a surface can be described in its principal co-ordinate frame (marked S' in Figure 4) by the matrix formulation,

$$\mathbf{x}'^{\mathrm{T}}\mathbf{A}'\mathbf{x}' = 1 \tag{2}$$

where $\mathbf{x}'$ represents a general three dimensional vector in S', and the matrix $\mathbf{A}'$ has components,

$$\mathbf{A}' = \begin{pmatrix} \frac{1}{a^2} & 0 & 0 \\ 0 & \frac{1}{a^2} & 0 \\ 0 & 0 & \frac{1}{b^2} \end{pmatrix} \tag{3}$$

where a and b are are the major and minor axes as shown in Figure 4. We derive these quantities from the path length travelled by the reflected pulse, l, as defined in Figure 4. Using simple geometrical arguments, we find:

$$\begin{aligned} a &= \frac{1}{2}\sqrt{l^2 - |\mathbf{t}-\mathbf{r}|^2} \\ b &= \frac{l}{2} \end{aligned} \tag{4}$$

To describe this in co-ordinate frame S, our 'real world' frame, we apply a translation to make the origins coincident, and use Rodrigues' rotation formula [11] to calculate the relevant rotation matrix, $\mathbf{R}$, such that

$$\mathbf{x}' = \mathbf{R}(\mathbf{x} - (\mathbf{t} + \frac{1}{2}(\mathbf{r}-\mathbf{t}))) = \mathbf{R}(\mathbf{x}-\mathbf{u}) \tag{5}$$

By transforming a regular three dimensional grid in system S to S' we can use geometry to calculate the intersection of this prolate spheroid with the cells, forming an SRSV grid for arbitrary surfaces. However, in pervasive systems we are usually able to restrict our interest to a subset of surfaces such as vertical walls, near-vertical screens, horizontal tables, etc. Such restrictions allow refinement of the method. As an example, in this paper we will be primarily interested in vertical surfaces, for which we wish to restrict the locus of $\mathbf{P}$ to spheroid points with a normal lying in the horizontal plane of S. To see this, consider the function,

$$f(\mathbf{x}) = |\mathbf{x}-\mathbf{t}| + |\mathbf{x}-\mathbf{r}| \tag{6}$$

which represents the path length for $\mathbf{t}$ to $\mathbf{x}$ to $\mathbf{r}$. The quantity $\nabla f(\mathbf{x})$ will point in the direction of the sum of the unit vectors of $(\mathbf{t}-\mathbf{x})$ and $(\mathbf{r}-\mathbf{x})$, which bisects the angle between these two vectors. Thus, $\nabla f(\mathbf{x})$ points in the direction of the normal to the reflecting surface for a specular reflection from $\mathbf{t}$ to $\mathbf{r}$. The spheroid is defined by the relation

$$f(\mathbf{x}) = l \tag{7}$$

and hence $\nabla f(\mathbf{x})$ points in the direction of the normal to the spheroid at $\mathbf{x}$. i.e. The tangent plane at any point on the spheroid is the reflecting surface that reflects a signal from $\mathbf{t}$ to $\mathbf{r}$, and by symmetry the locus of $\mathbf{P}$ is then the intersection of the spheroid with a plane which passes through the origin of S'.

To describe this intersection, we define a further co-ordinate system, S'', which has the same origin as S', but with the $\mathbf{z}''$ direction normal to the intersection plane, and the $\mathbf{y}''$ direction perpendicular to this, but within the vertical plane containing both $\mathbf{r}$ and $\mathbf{t}$ (see Figure 4). We can link the frames S' and S'' by a rotation matrix, $\mathbf{R}'$, such that

$$\mathbf{x}'' = \mathbf{R}'\mathbf{x}' \tag{8}$$

To set up the axes in practice, we calculate a point of intersection ($\mathbf{P}$ in Figure 4) of the vertical plane through $\mathbf{r}$ and $\mathbf{t}$ with the spheroid, and with a horizontal normal. Then we calculate a point on the spheroid with $z' = 0$ and a horizontal normal. Combined with the S' origin, we then have three non-co-linear points; sufficient to derive the plane normal, $\mathbf{n}'$. When $\mathbf{n}'$ is normalised, (8) implies that

$$\mathbf{R}'\mathbf{n}' = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{9}$$

We choose $\boldsymbol{y}''$ to point from the origin to the point $\mathbf{P}$, which implicitly defines the direction of $\mathbf{x}'$ in the Cartesian S' frame. We can then relate the frames S' and S'' by a rotation matrix $\mathbf{R}'$ and assert,

$$\mathbf{x}''^{\mathrm{T}}\mathbf{R}'\mathbf{A}'\mathbf{R}'^{\mathrm{T}}\mathbf{x}'' = \mathbf{x}''^{\mathrm{T}}\mathbf{A}''\mathbf{x}'' = 1 \tag{10}$$

which, referring back to (2), defines an elliptical intersection in the $z'' = 0$ plane.

Once the ellipse parameters have been calculated, we can perform SRSV by transforming the vertices of SRSV cells to that co-ordinate system, and searching for interceptions of the faces with the ellipse.

3.2 Using The Spheroid Normals

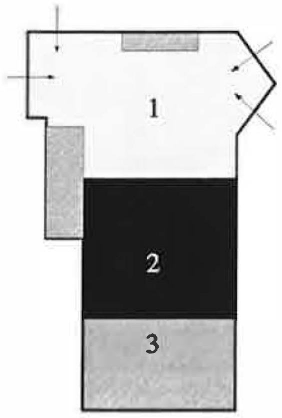
We can extend the basic premise of SRSV by accounting for surface normals, vastly simplifying any subsequent analysis. When a spheroid intersects a cell, there is an associated average normal to the spheroid within that cell. If there is to be a reflecting surface giving rise to that spheroid within that cell, it must have a normal direction parallel to the spheroid normal. Thus, we can associate each SRSV vote within a cell with a particular three dimensional normal direction. In practice, we associate a series of angular bins with each cell, and add votes to the angular bin that contains the average spheroid normal within that cell. If a planar reflecting surface exists within that cell, the SRSV count for the angular bin containing its normal direction should vastly exceed the counts in the remaining angular bins.

When considering vertical walls, we need only associate cells with angular bins in the horizontal plane. We calculate the average normal to the ellipse within each cell it intersects, project that direction into the horizontal plane and increment the relevant angular bin for the cell. This is a particularly useful representation when searching for walls, since we are really searching for contiguous, vertical, planar surfaces that extend across multiple cells, all with high SRSV counts in the angular bin corresponding to the surface normal.

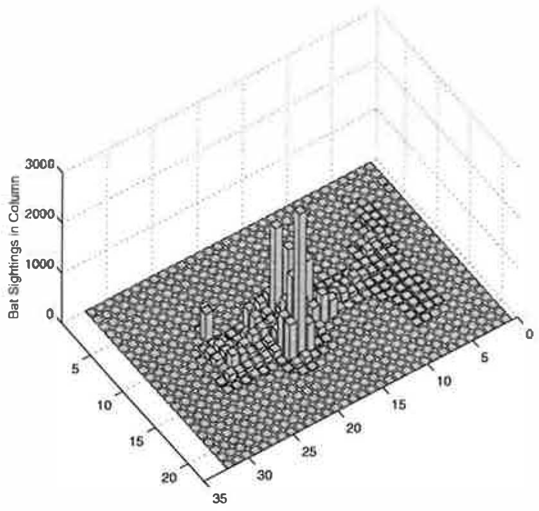
4 Implementation and Results

Experimental validation was performed by post-processing real reflections collected from the Bat system. We have logged the reflections, caught in a communal coffee area, for Bats worn by personnel for a period of two days, producing a distribution of sightings and reflections typical for an area in daily use. Figure 5 illustrates the shape of the area and any fixed objects (hatched) and identifies three key Regions within it (labelled 1,2,3). Region 1 is used as a through-way, with people rarely stopping. Consequently, we see a low number of sightings here. Region 2 is a general communal area, where people tend to congregate and remain for extended periods of time, building up large numbers of sightings. Region 3 is an area with a low density of Bat receivers in the ceiling, and a high density of ceiling obstructions. Relative to Regions 1 and 2, sightings are rare here.

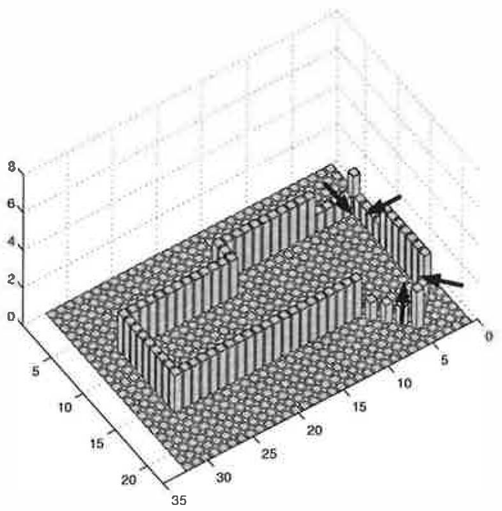
Figure 5(b) shows the actual distribution of sightings collected during the test period. This data was processed using an SRSV cell size of 0.2m, using 6897 collected reflections. The initial result is shown in Figure 6; the SRSV counts shown are the sums of each vertical column of SRSV cells. This 'collapsed column' representation helps to capture the fact that the walls are vertical and thus all cells in a vertical column provide evidence for a wall at that position. The highly non-uniform usage of the area by personnel is reflected in the results. We see a high density of intersections in Region 2, a direct result of users remaining in roughly the same place for many sightings, building up intersections in that



(a) A schematic view of the communal coffee area



(b) The distribution of sightings collected during testing



(c) A representation of the room walls using the same collapsed column grid as (b). The arrows show door positions.

Figure 5: The testing area

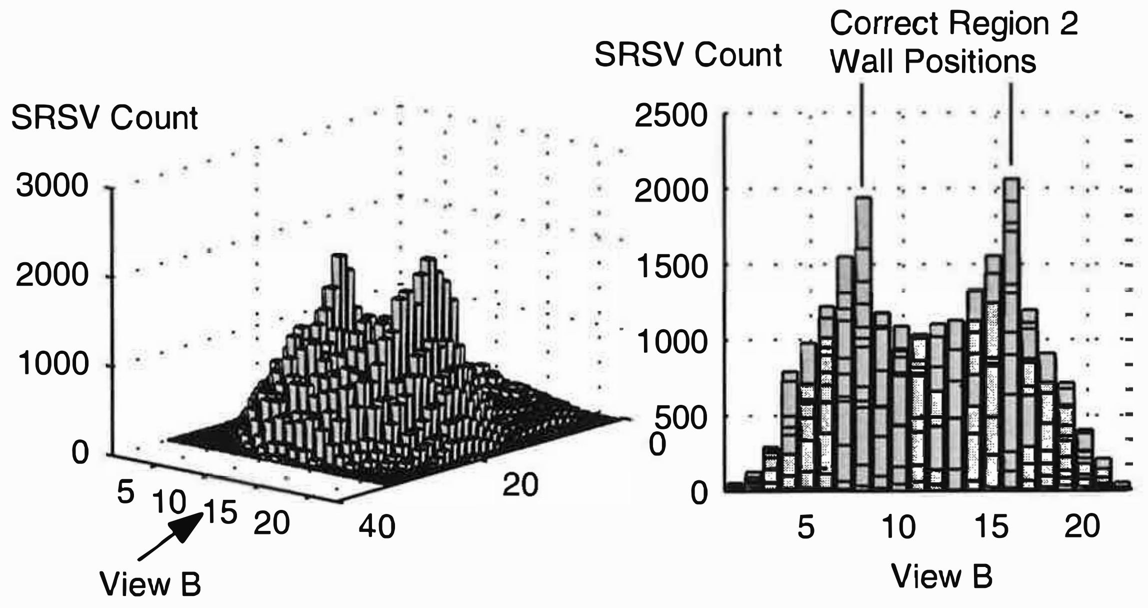


Figure 6: The column-collapsed SRSV grid for the coffee area from a series of viewing angles

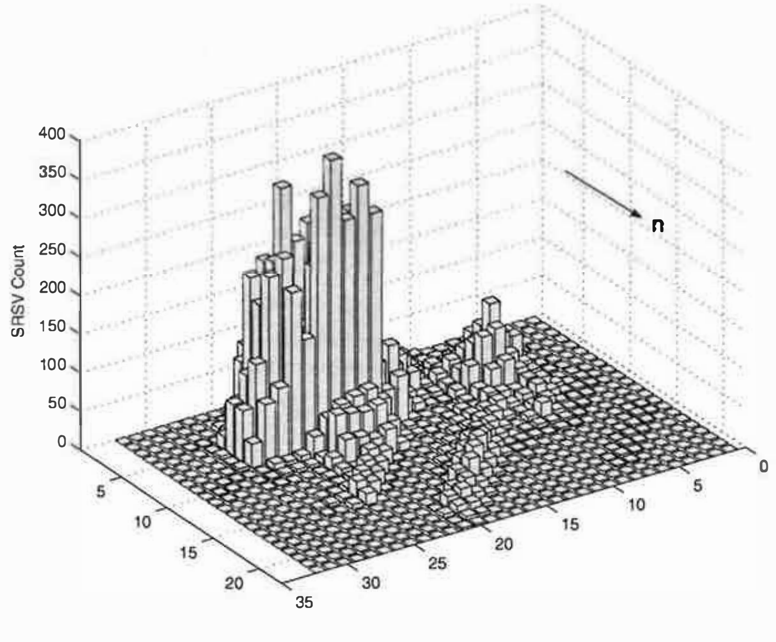
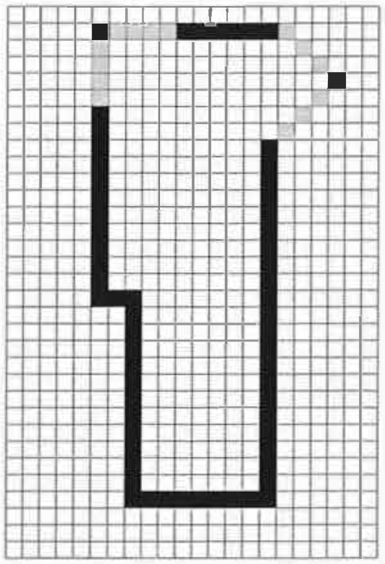


Figure 7: The SRSV count for angular bins containing direction $\mathbf{n} \pm 10°$.


(a) Correct representation of the test area (grey cells represent doors).

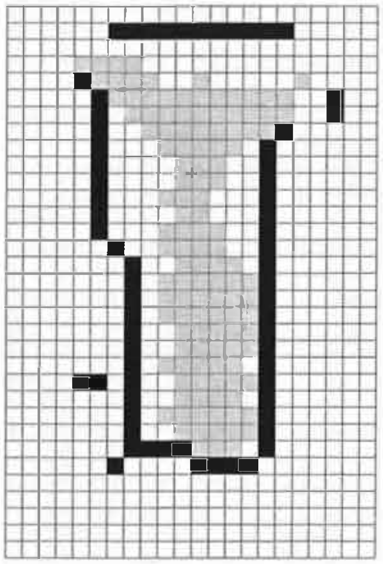

(b) Results of SRSV (grey cells show the eroded sighting distribution)

Figure 8: SRSV Results

Region. There are strong maxima in the SRSV counts in Region 2, which correctly trace out the walls on either side. Note that the SRSV count inside this Region is non-zero, increasing with the number of sightings, despite the fact that there can be no wall there. This potentially misleading situation illustrates the need to store the SRSV normal directions in angular bins. Figure 7 shows the same data, but for a specific angular bin of width 20° containing the wall normal direction. We see that the SRSV counts within Region 2 are drastically reduced, since the high counts in Figure 6 are distributed across the angular bins in the SRSV cell.

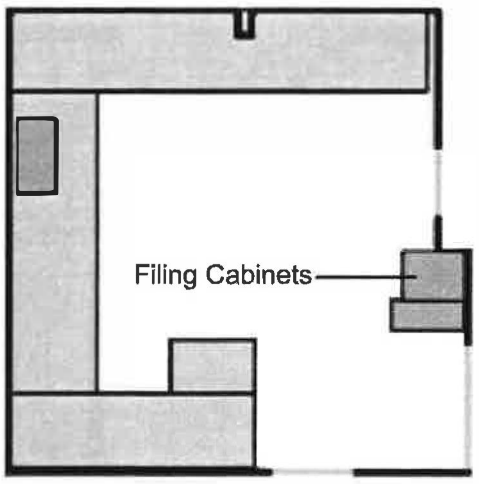
We can further improve each angular view by excluding cells in a column that has seen a sighting of a Bat (thereby implying no wall can be there). In practice, to account for positioning inaccuracies and the small possibility of a wall and a Bat coexisting within a cell of finite size, we have found it best to erode the area of sightings by one cell in every direction. This leaves us with a core shape of sightings that we are confident covers cells that do not contain walls. To encapsulate this fact, we zero the SRSV count for any column within which a Bat was sighted.

To extract the wall positions, we applied the following simple algorithm for the four normal directions that follow the axes of the SRSV grid:

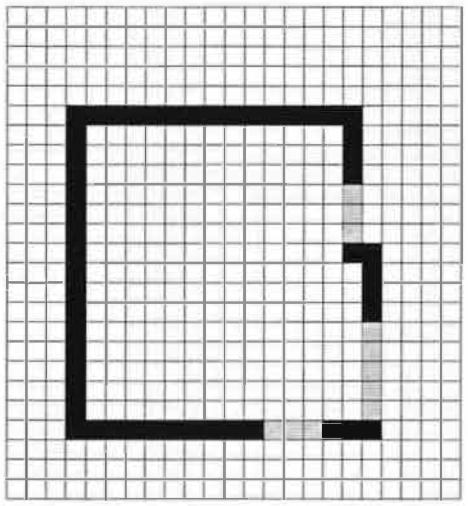
1. Create the SRSV grid for the angular bin containing the normal (e.g. Figure 7) of interest.
2. Set SRSV counts to zero for cells contained in the eroded region of sightings
3. Scan the grid perpendicular to the normal direction, amalgamating cells with non-zero SRSV counts into single 'walls', allowing a maximum gap of two cells (0.4m) within each. This gap allows for wall continuity across regions of low or no occupancy.
4. Calculate the average SRSV count for each wall, and assign the value to each cell within the wall.
5. Scan the grid parallel to the normal direction, taking the cell with the maximum average SRSV count as containing a wall.

The results of applying this algorithm are presented in Figure 8, and are encouraging.

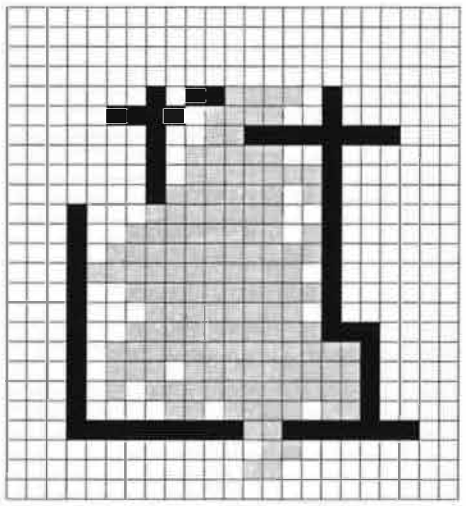
The position and orientation of the longer side walls has been correctly extracted, with the overall room shape being determined to a good approximation. Despite the vastly different uses of



(a) A graphical representation of the room and its furniture



(b) Correct Representation of the room on the SRSV grid



(c) Results of SRSV (grey cells show eroded sighting distribution)

Figure 9: The results of applying SRSV to another room.

Regions 1 and 2, we are still able to extract the wall positions with good accuracy. The door positions, shown in Figure 8(a), are clearly present, with the exception of the top-most door. This door leads to an office that was not in use during testing, and hence the door remained closed, completing the wall.

Importantly, the method is clearly superior to simply using the sighting distribution to determine the room shape, as shown by the shaded region of Figure 8(b). The SRSV method is capable of inferring wall positions in areas penetrated by ultrasonic signals, but where Bats are not sighted. This is a useful feature in office environments, where desks and machines typically obstruct users from entering areas adjacent to walls.

The extraction algorithm used works well when we specify the normal axes to examine. However, the vast majority of office buildings contain rectangular rooms, so such a scenario is not contrived. The principal axes may be entered by hand, or we may make the assumption that once the orientation of one room or wall is determined, we give precedence to the same normal direction (and those parallel and perpendicular to it) throughout the building. Determination of a single wall orientation is possible by simply looking for a long sequence of cells with locally high SRSV counts in angular bins consistent with the sequence normal, or by using more complex image techniques such as a Hough Transform [8].

For comparison, Figure 9 gives a representation of another room within our Laboratory, and the results of applying SRSV analysis to sightings recorded within it over a period of three days. Here, we find the results to be similarly encouraging. The absence of a colleague who sits in the top left corner of 9(a) during the testing period resulted in poor determination of walls in that area, as expected. However, the remaining wall positions were correctly determined. In particular, it is interesting to note that the filing cabinets marked in 9(a) were treated as an extension of the wall nearby due to the large, smooth and vertical surface they presented.

5 Data Evolution

To demonstrate the evolution of SRSV data as sightings are collected, a further dataset was collected from within a long, straight corridor. This gave a further opportunity to test the method, and a very strong feature to examine, in the form of one of the walls.

A total of 4000 sightings were analysed, collected over a period of one day. Figure 10 shows a series of views of the collapsed grid count for a normal direction perpendicular to one of the walls after different numbers of sightings, n. The correct wall position lies within the bin labelled 10 in the graphs. The evolution of a peak within this bin is clearly evident.

It is important to realise that we cannot accurately define the number of sightings required to extract a feature. There is no guaranteed number of single reflections associated with each sighting, and furthermore no guarantee that the sightings are distributed in such a way as to produce reflections along the entire length of the feature. This can be observed in Figure 10 by the apparent lack in difference between 10(f) and 10(g). Nevertheless, the graphs shown in Figure 10 serve to illustrate a typical evolution.

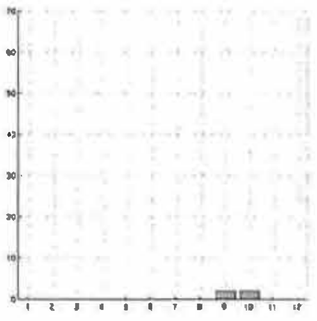
6 Further Applications of SRSV

SRSV offers the opportunity to detect static and dynamic details of great use to the pervasive computing community. Whilst in this paper we have applied SRSV analysis in the context of static vertical walls, the concept is not limited to them. We can also detect other vertical features, such as dynamic walls (temporary partitions), door positions and states (open or closed), and positions and orientations of monitors (particularly useful in the context of hot-desking).

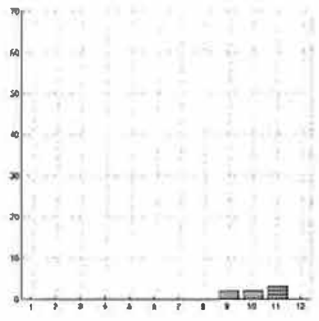
Note that the analysis algorithm for the SRSV grid may need to be tailored to extract features of limited extent, such as monitors. The algorithm presented above searches for long, contiguous lines within the collapsed grid, which we would not expect to observe for monitors. Instead, we would search for well-defined, bounded planes of normals at heights useful for displays.

It is possible to apply the same ideas to non-vertical surfaces by a straightforward coordinate rotation. In the specific case of the Bat system, however, there is little information to be gained about non-vertical surfaces. This is due to the directional nature of the ultrasonic emission from a Bat.

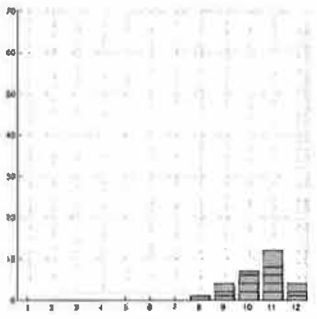
Since the Bat receivers are designed to be ceiling mounted, the ultrasonic emission from Bats is designed to be directed primarily upward, toward the ceiling. Furthermore, for most of time Bats



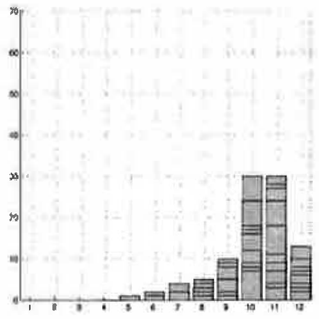
(a) n=10



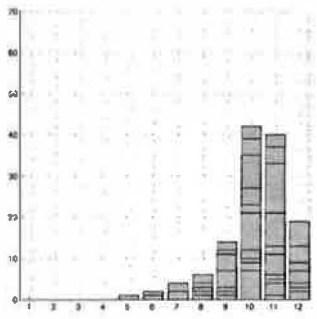
(b) n=100



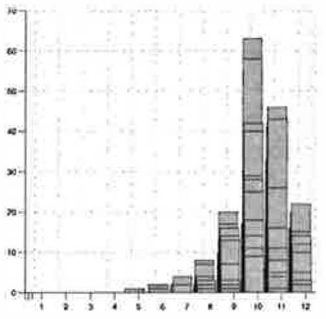
(c) n=500



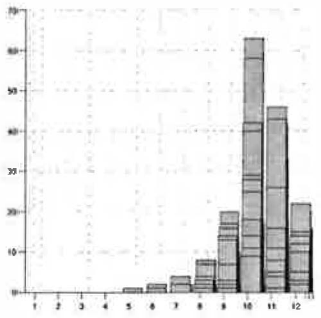
(d) n=1000



(e) n=1500



(f) n=2500



(g) n=4000

Figure 10: Evolution of an SRSV grid. The graphs show a view along the length of a corridor. A wall is located within horizontal bin 10

are worn by personnel at heights consistent with the characteristic standing and sitting heights of those people. Non-vertical surfaces, such as tables, are consequently below the usual height of the Bat, and it is unlikely that a signal from the Bat will reflect from them and reach the receivers.

Use of the method to find only vertical surfaces with Bat system data does not imply any limitation of its use with other positioning systems. For example, signals in proposed UWB positioning systems would be expected to reflect extensively from surfaces of all orientations within the environment.

7 Conclusions and Further Work

We have presented a novel method for surface discovery using reflections. Our results indicate that it can potentially provide a rich source of world information. Our implementation has centred around wall detection, which we have demonstrated with good results based on a relatively small data set. The method in general, however, is easily extended to search for non-vertical reflecting surfaces, by using intersection of prolate spheroids and SRSV cells which record intersection normals in three dimensions.

We envisage this data source being combined with other sources (many of which are discarded by today's positioning systems) to provide robust information about the environment. In particular, we hope to use these, and related, techniques to determine the position and orientation of large flat-screen displays - a common source of problems for ultrasonic positioning which we intend to compensate for. Furthermore, the usage of archived personnel positions can be used to derive complementary environmental information, such as region connectivity and the position of large scale objects [9].

Whilst the results presented here are for an ultrasound-based system, the theory applies equally to any system that can resolve multipathed effects. In particular, the methods presented here should transfer directly to Ultra-Wideband (UWB) radio positioning systems, which have the intrinsic capability of resolving and timing multipathed signals [7].

Beyond the niche of pervasive computing, the method can be applied to autonomous navigation and map building. In these fields, we typically use a variety of ranging and signal bouncing techniques to permit a robot to 'learn' its environment using probabilistic methods [15]. SRSV here could provide a useful extra information source that improves the probability calculations. Furthermore, such a robot can be programmed to adopt a motion pattern that targeted specific areas of uncertainty within an environment.

8 Acknowledgements

The authors are grateful to both the EPSRC and AT&T for supporting this work through a research grant and CASE award respectively.

References

[1] M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, and A. Hopper. Implementing a sentient computing system. *IEEE Computer*, 34(8), August 2001.

[2] P. Bahl and V. N. Padmanabhan. RADAR: An in-building RF-based user location and tracking system. In *INFOCOM 2*, pages 775–774, 2000.

[3] P. Bahl, V. N. Padmanabhan, and A. Balachandran. Enhancements to the RADAR user location and tracking system. Technical report, Microsoft Research, February 2000.

[4] B. Barshan. Ultrasonic surface profile determination by spatial voting. In *Proceedings of the IEEE Instrumentation and Measurement Technology Conference, Budapest, Hungary*, pages 583–588, May 2001.

[5] B. Barshan and D. Baskent. Comparison of two methods of surface profile extraction from multiple ultrasonic range measurements. *Meas. Sci. Technology*, 11:833–844, April 2000.

[6] D. Bohn. Environmental effects on the speed of sound. *Journal of the Audio Engineering Society*, 36(4):223–231, April 1988.

[7] Time Domain. PulsOn technology overview. Technical report, Time Domain Corporation, 2001. http://www.timedomain.com.

[8] R. O. Duda and P. E. Hart. Use of the hough transform to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.

[9] R.K. Harle and A. Hopper. Using Personnel Movements For Indoor Autonomous Environment Discovery. To appear in PerCom 2003, 2003.

[10] J. Hightower and G. Borriello. Location sensing techniques. *IEEE Computer*, August 2001.

[11] R. M. Murray, Z. Li, and S. S. Sastroy. *A Mathematical Introduction to Robotic Manipulation*. CRC Press, 1994.

[12] N. B. Priyantha and A. Chakraborty and H. Balakrishnan. The Cricket Location-Support System. *Proceedings of the Sixth Annual ACM International Conference on Mobile Computing Networking*, August 2000.

[13] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. CUP, 1992.

[14] N. B. Priyantha, K. L. M. Allen, H. Balakrishnan, and S. J. Teller. The cricket compass for context-aware mobile applications. In *Mobile Computing and Networking*, pages 1–14, 2001.

[15] S. Thrun. Learning maps for indoor mobile robot navigation. *Artificial Intelligence*, 99(1):21–71, 1998.

[16] A. M. R. Ward. *Sensor-driven Computing*. PhD thesis, Cambridge University, August 1998.

The Lighthouse Location System for Smart Dust[1]

Kay Römer
Department of Computer Science
ETH Zurich, Switzerland
roemer@inf.ethz.ch

Abstract

Smart Dust sensor networks – consisting of cubic millimeter scale sensor nodes capable of limited computation, sensing, and passive optical communication with a base station – are envisioned to fulfill complex large scale monitoring tasks in a wide variety of application areas. In many potential Smart Dust applications such as object detection and tracking, fine-grained node localization plays a key role. However, due to the unique characteristics of Smart Dust, traditional localization systems cannot be used. In this paper we present and analyse the Lighthouse location systems, a novel laser-based location system for Smart Dust, which allows tiny dust nodes to autonomously estimate their location with high accuracy without additional infrastructure components besides a modified base station device. Using an early 2D prototype of the system, node locations could be estimated with an average accuracy of about 2 % and an average standard deviation of about 0.7% of the node's distance to the base station.

1 Introduction

Wireless sensor networks (WSN) [1] are currently an active field of research. A WSN consists of large numbers of cooperating small-scale nodes capable of limited computation, wireless communication, and sensing. In a wide variety of application areas including geophysical monitoring, precision agriculture, habitat monitoring, transportation, military systems and business processes, WSNs are envisioned to fulfill complex monitoring tasks.

In many typical sensor network applications, fine-grained physical locations of individual sensor nodes play an important role. Examples include target detection (*where* is the target?), target tracking (*where* and *how fast* is a target moving?), and target classification (what are *size* and *shape* of the target?). Moreover, *location-dependent queries* can increase both the utility and the lifetime of a sensor network. By directing a query only to nodes in a certain geographical region or by making certain query parameters (e.g., sensor sampling rate) a function of the node location, valuable energy resources can be saved by restricting the sensor network activity to what is actually needed to answer a query.

Techniques for physical location sensing have been studied for a long time, among others, in the context of mobile computing systems [16]. More recently, some of the approaches developed there have been adopted for WSN [3, 4, 6, 10, 13, 25], mainly focusing on systems based on certain characteristics such as time-of-flight, received signal strength, signal range of ultrasound and radio waves. This adoption is often possible, because in many respects a wireless sensor node is not too much different from a mobile computing device like a PDA with WLAN access. Compared to a PDA, however, sensor nodes have rather limited resources due to their required small size and cost. Nevertheless it is often possible (both energy-wise and size-wise) to equip such sensor nodes with low power radios or small ultrasound transducers as enablers for location sensing systems.

However, research is already on the way to create the next generation of sensor nodes, for example at UC Berkeley [19, 31]. Due to their envisioned cubic-millimeter size, they are called "Smart Dust". By making nodes inexpensive and easy-to-deploy, Smart Dust opens up new applications areas. The radical size reduction mandates a revolutionary change in the used communication technology when compared to current WSN technology. Traditional radio technology presents a problem because Smart Dust nodes offer very limited space for antennas. Furthermore, radio transceivers are relatively complex circuits, making it difficult to reduce their power consumption to the level required by Smart Dust. In order to meet these requirements, [19] suggests the use of laser-based free-space optical transmission. However, due to power restrictions, near future Dust nodes will most likely make use of passive optical communication only, limiting communication to a bidirectional link between a base station device and each node.

[1]The work presented in this paper was supported in part by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

This revolutionary new technology presents a whole new set of challenges to location sensing systems. Traditional systems based on radio waves and ultrasound are ruled out due to their power consumption and size requirements. The expected unprecedented scale of Smart Dust deployments will further challenge a location system.

In this paper, we present the Lighthouse location system for future WSN systems that are similar to the early Smart Dust prototypes developed at UC Berkeley [19]. By extending the base station, this system allows Smart Dust to autonomously estimate their physical location with respect to a base station with high precision over distances of tens of meters without node calibration. Besides a modified base station, the system does not require any additional infrastructure components. This is achieved by a new cylindrical lateration method. In contrast to traditional spherical methods, this approach does not have a wide baseline requirement (see Section 3). On the receiver side, only a simple optical receiver (amplified photo diode), moderate processing capabilities, and little memory are needed. That is, only marginal changes to the Smart Dust prototype developed at UC Berkeley are necessary.

We first describe future Smart Dust systems and compare them to more traditional commercial-off-the-shelf (COTS) sensor nodes. We will then describe the challenges of a location system for Smart Dust, before presenting the Lighthouse location system itself. The latter includes a description of the basic approach, the presentation of a prototype system, a set of initial measurements, and a first analysis of several system aspects. We conclude the paper with mentioning related work, our current work, and future research directions.

2 Smart Dust

As described in [19], Smart Dust nodes as envisioned by the Berkeley Smart Dust project will consist of a small battery, a solar cell, a power capacitor, sensors, a processing unit, an optical receiver, and a corner-cube retroreflector (CCR) within a space of about one cubic millimeter. Later versions might also contain an active transmitter based on a semiconductor laser diode. However, the high power consumption of the laser diode significantly limits the usefulness of such a component. Therefore, in the near future, communication will be possible only between sensor nodes and a so-called base station transceiver (BST).

The BST mainly consists of a steerable laser and a compact imaging receiver. For downlink communication, the BST points the modulated laser beam at the optical receiver of a node. For uplink communication, the BST points an unmodulated laser beam at the node, which modulates the laser beam and reflects it back to the BST using its CCR. Using its imaging receiver, the BST can receive and decode transmissions from dust nodes.

Obviously, this communication scheme requires an uninterrupted line-of-sight path. For many of the environmental monitoring applications envisioned for Smart Dust, however, this is not a major problem. Additionally, communication is only possible if the node's optical receiver and CCR point to the BST, so that only a fraction of deployed nodes will be able to communicate. This should not be a problem, however, if the dust node density is high enough.

Having these Smart Dust characteristics in mind, what are the differences to state-of-the-art RF-based WSN with respect to location sensing? The main differences clearly stem from the tremendous size reduction from several cubic centimeters to a few cubic millimeters. The small size also imposes tight limits on the available energy, which in turn restricts communication, memory, and processing capabilities of dust nodes. Another difference is caused by the passive optical communication scheme of dust nodes, making near future Smart Dust systems essentially single-hop networks without direct node-to-node communication. We can summarize the differences between current WSN and future Smart Dust systems as follows with respect to location systems:

- Small size: current RF antennas for radio waves and transducers for ultrasound are too large for dust nodes.
- Mobility: future Smart Dust nodes are likely to be small enough to be moved by winds or even to remain suspended in air, buoyed by air currents. This is in contrast to current sensor nodes, which are typically immobile due to their size and weight.
- Large scale: the expected small size and low cost of future Smart Dust nodes will allow very large scale deployments in terms of the number of nodes.
- Limited energy: the power consumption of current RF transceivers, for example, is too high for dust nodes.
- Limited computing and memory resources: many wide-band ultrasound location systems, for example, sample at more than 40kHz and do signal processing on the sampled data, resulting in large memory and processing overheads [13], which is not possible on a Smart Dust node.
- Single-hop network topology: current WSN location systems often assume multi-hop networks, where a node can cooperate with its neighbors in order to compute its location [3, 25], which is likely not true in near-future Smart Dust systems.

3 Localization Challenges for Smart Dust

In this section we examine the challenges a Smart Dust system as outlined in the previous section presents for a location system.

3.1 Accuracy

The accuracy of the physical location estimates required by a Smart Dust deployment very much depends on the sensed phenomenon and the accuracy requirements of the application. In tracking applications, for example, a location grain size of about the size of the monitored phenomenon is often sufficient. That is, the required accuracy of the location system is fine-grained and ranges from centimeters (for tracking a flying insect) to meters (for tracking a large animal herd).

Note that in a typical deployment, the distance between the base station and dust nodes is in the order of tens of meters. A location system for Smart Dust should provide the desired degree of accuracy even under this condition.

3.2 Localized Location Computation

For certain applications, Smart Dust nodes need to know their own physical locations. In the following we will point out reasons for localized location computation.

In a typical application, dust nodes sample environmental parameters by reading attached sensors at regular intervals. The obtained time series of sensor readings are then preprocessed in some application-specific way before sending off relevant data to the base station.

It is a well-known observation from statistical data management that areas where changes are happening most rapidly (hot spots) should be sampled at a higher rate [11]. On the other hand, the sampling rate should be as small as possible to save energy. *Location-dependent queries* offer a solution for this tradeoff by making certain query parameters – such as the sensor sampling rate – a function of node location. A location-dependent query can be sent to the whole Smart Dust network with a single (logical) broadcast. Nodes have to obey the query parameters according to their (mutable) current location.

In order to save scarce communication bandwidth and energy, dust nodes typically cannot report detailed time series of sensor readings to the base station [11]. Instead, nodes preprocess such time series locally [28] in order to come up with a smaller and more high-level data representation (e.g., histogram or distribution function), which is then sent to the base station rather infrequently. For many applications (e.g., monitoring the spatial distribution of air pollution), preprocessing depends on the (mutable) physical location at which individual sensor readings are obtained.

In many traditional location systems such as [30], an external infrastructure observes objects and computes their location. This approach moves the burden of location computation from the nodes to a more powerful infrastructure. In Smart Dust applications where nodes need to know their location, however, the base station would have to send an individual location update message to each node of the network one by one. That is, the associated communication overhead grows linearly with the number of nodes. The following example shows that sending location updates to a network of 1000 mobile dust nodes every 20 seconds can hog all the communication bandwidth.

Sending a location update to a node involves aiming the laser beam at the node and sending a location update message. Aiming the beam typically involves aligning a steerable MEMS mirror which operates at a few hundred Hertz [21]. We will assume a mirror bandwidth of 100 Hz, a downlink communication bandwidth of 10 kbit per second, and an update message size of 20 bytes or 160 bits (3*4 bytes for physical location, plus node addressing and protocol overhead). With these parameters, the base station can send a location update to a single node about every 0.02s. Sending location updates to all nodes of a network with 1000 mobile nodes will then take 20 seconds.

Another reason for localized location computation is privacy. If dust nodes are attached to people, places or things, knowing the location of the node would also disclose the location of its host to the infrastructure. This, however, is valuable information in many cases, which can be easily abused for recording the behavior of people [20, 24]. Therefore, whenever possible, it is favorable to compute locations in the nodes themselves without disclosing them to a potentially untrusted infrastructure.

3.3 Low Cost

A location sensing system for Smart Dust imposes certain space, capital, and time costs [16]. These are due to software and hardware required for location sensing on the dust nodes and in the infrastructure (i.e., the base station). Space costs involve the amount of installed infrastructure and the node hardware's size and form factor. Capital costs include factors such as the additional price per Dust node and base station. Time costs include the overhead for system installation, calibration, and administration.

The envisioned application areas for Smart Dust impose certain limits on these costs. The intended low capital cost and small size, for example, require that the location sensing hardware overhead needed on the nodes is minimal. Ideally, a location system would reuse the existing optical receiver instead of adding additional hardware to the nodes. On the other hand, adding additional hardware to the base station is not so critical, because there will be very few base stations

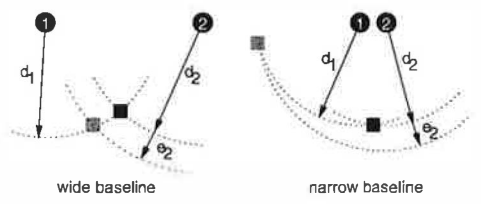


Figure 1: Errors in estimated node location depend on whether or not the points of reference for multilateration form a wide baseline.

when compared to the number of deployed dust nodes. However, introducing additional infrastructure components is not a good idea, because installation and administration of the latter contradicts the *ad hoc* nature of sensor networks.

Note that the limitation to a single piece of infrastructure (i.e., the base station) is a challenging one. In the Smart Dust single-hop network, where nodes cannot communicate directly with each other, node localization requires an external infrastructure. In multilateration-based systems, for example, the distances $d_1, ..., d_N$ to multiple points of reference $1, ..., N$ provided by the infrastructure are measured and used to compute the node's location. In order to achieve high accuracy, the reference points should form a *wide baseline*, that is, the distances among the reference points should be in the order of the distance of the node to the reference points. Figure 1 illustrates this situation in 2D. There, the distances d_1 and d_2 of the node to the two reference points 1 and 2 are measured. The node's location is computed as the intersection point of two circles with radius d_1 and d_2 centered at the reference points. If the two reference points form a wide baseline, an error e_2 in the distance measurement d_2 causes only a small error in the estimated node location. If the two reference points are close together, the same error e_2 causes a large error in the estimated node position.

Implementing a wide baseline typically requires multiple geographically distinct infrastructure components in order to provide the reference points (*beacons*). Moreover, placement of the beacons is often a non-trivial problem [5]. Usually, the exact locations of the reference points have to be known in order to compute node locations [4, 15, 24, 30]. In some systems, the beacons even need accurately synchronized clocks [18]. In order to avoid these problems, we developed a new localization approach based on cylindrical lateration, which does not have a wide baseline requirement.

Another important overhead involved in setting up a localization system is node calibration [32] in order to enforce a correct mapping of sensor readings to location estimates. In systems based on RF received signal strength (RSSI), for example, the received signal strength is mapped to a range estimate. Variations in transmit power and frequency among the nodes can cause significant inaccuracies in the range estimates when used without calibration [17]. Since the cheap low-power hardware used in WSN typically introduces a high variability between nodes, sensor nodes have to be individually calibrated. This, however, may not be feasible in Smart Dust installations due to their expected large scale. The Lighthouse location system does not require node calibration and thus completely eliminates the overhead of the latter.

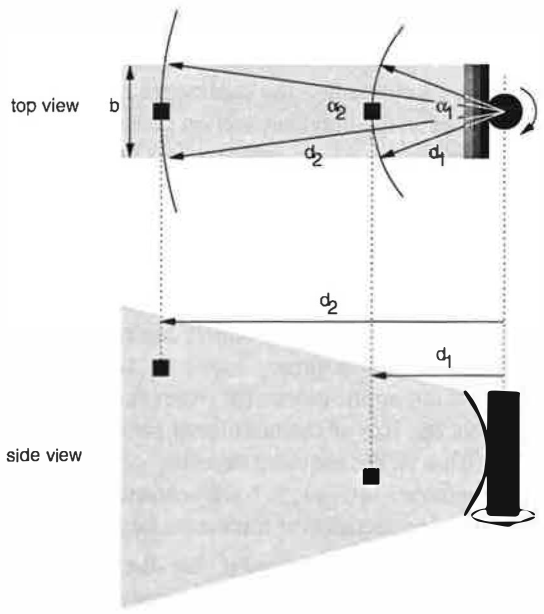


Figure 2: Top and side view of an idealistic lighthouse with a parallel beam of light.

4 The Lighthouse Location System

This section presents the Lighthouse location system for Smart Dust. In order to point out the basic ideas behind this system, we will first examine a simplified idealistic system. This examination will be followed by a more thorough discussion of a realistic system that can actually be built. We will go on by presenting a first prototype implementation, an initial set of measurements, and a first analysis of several aspects of the system.

4.1 An Idealistic System

Consider the special lighthouse depicted in Figure 2, which has the property that the emitted beam of light is parallel (i.e., has a constant width) with width b when seen from top. When seen from the side, the angle of beam spread of the parallel beam is large enough so that it can be seen from most points in space.

When this parallel beam passes by an observer, he will see the lighthouse flash for a certain period of time t_{beam}. Note that

t_{beam} depends on the observer's distance d from the rotation axis of the lighthouse since the beam is parallel. Assuming the lighthouse takes t_{turn} for a complete rotation, we can express the angle α, under which the observer sees the beam of light as follows:

$$\alpha = 2\pi \frac{t_{\text{beam}}}{t_{\text{turn}}} \tag{1}$$

Figure 2 shows two observers (depicted as squares) at distances d_1 and d_2 and the respective angles α_1 and α_2. Now we can express d in terms of α and the width b of the beam as follows:

$$d = \frac{b}{2\sin(\alpha/2)} \tag{2}$$

By combining Equations 1 and 2 we obtain the following formula for d in terms of b, t_{beam}, and t_{turn}:

$$d = \frac{b}{2\sin(\pi t_{\text{beam}}/t_{\text{turn}})} \tag{3}$$

Note that the distance d obtained this way is the distance of the observer to the lighthouse rotation axis as depicted in the side view in Figure 2. That is, all the points in space with distance d form a cylinder (not a sphere!) with radius d centered at the lighthouse rotation axis.

Based on the above observations, we can build a simple ranging system consisting of a lighthouse and an observer. The observer device contains a photo detector and a clock. When the photo detector first sees the light it records the corresponding point in time t_1. When the photo detector no longer sees the light it records t_2. When it sees the light again it records t_3. With $t_{\text{beam}} := t_2 - t_1$ and $t_{\text{turn}} := t_3 - t_1$ the observer can apply Equation 3 in order to calculate its distance d from the lighthouse rotation axis. Note that if t_{turn} is constant it has to be measured only once since it does not change with distance. Also note that the necessary hardware resources of the observer device are matched by a Smart Dust node as explained in Section 2.

This ranging scheme can be used to build a *single device*, which allows observers to autonomously determine their position relative to it in three dimensional space. This device consist of three lighthouses with mutually perpendicular rotation axes as depicted in Figure 3. Assuming an observer measures the distances d_x, d_y, and d_z as indicated above, its location can be determined by computing the intersection point(s) of three cylinders with radius d_x, d_y, d_z centered at the respective lighthouse rotation axes. Note that there are 8 such intersection points in general, one in each of the 8 quadrants of the coordinate system. If we can ensure, however, that all observers are located in a single quadrant (e.g., the main quadrant defined by the points (h_x, h_y, h_z) with $h_x, h_y, h_z \geq 0$), there is a unique intersection point. This

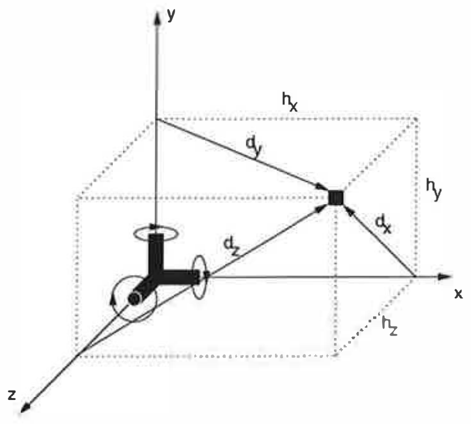


Figure 3: 3D Localization support device consisting of three mutually perpendicular lighthouses.

intersection point can be obtained by solving the following equation system for h_x, h_y, h_z:

$$\begin{aligned} d_x^2 &= h_y^2 + h_z^2 \\ d_y^2 &= h_x^2 + h_z^2 \\ d_z^2 &= h_x^2 + h_y^2 \end{aligned} \tag{4}$$

Note that this equation system does not necessarily have a solution, since the values d_x, d_y, d_z are only approximations obtained by measurements. If there is no solution, an approximation for the intersection point can be obtained using minimum mean square error (MMSE) methods. The solution (h_x, h_y, h_z) obtained this way minimizes the sum of the squares of the differences of the left hand and right hand sides of the equations 4. However, if the equation system has a solution, it can be directly solved using the following set of equations, again assuming that the observer is located in the main quadrant of the coordinate system depicted in Figure 3:

$$\begin{aligned} h_x &= \sqrt{(-d_x^2 + d_y^2 + d_z^2)/2} \\ h_y &= \sqrt{(d_x^2 - d_y^2 + d_z^2)/2} \\ h_z &= \sqrt{(d_x^2 + d_y^2 - d_z^2)/2} \end{aligned} \tag{5}$$

The setup of the complete location system can now be described. The base station is equipped with three mutually perpendicular lighthouses as depicted in Figure 3. At startup, the base station broadcasts certain calibration parameters (e.g., the beam width b for each of the lighthouses) to all dust nodes. The latter use a real-time clock to measure the amount of time during which each of the lighthouses beams are visible. Using Equations 3 and 4, nodes can autonomously compute their location in the reference grid defined by the base station's three lighthouses.

The description of the system's principles gives rise to a number of practical questions. First of all, it is not clear at all

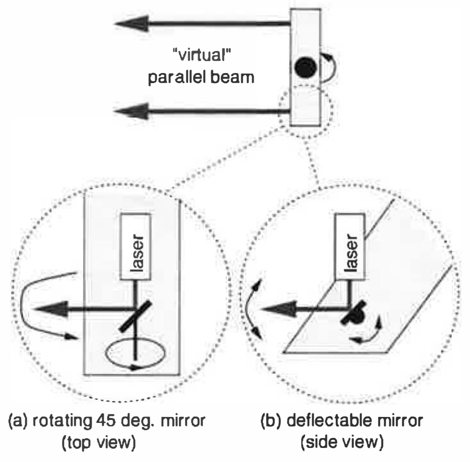


Figure 4: A rotating lighthouse with a "virtual" parallel beam whose outline is defined by two parallel laser beams. Rotating (a) or deflectable mirrors (b) are used make the laser beams scan the northern hemisphere of the lighthouse.

whether a system fulfilling the above requirements (e.g., parallel beam) can actually be built in practice. Moreover, we did not discuss the problem how a dust node can distinguish the different beams of the lighthouses, or what happens if a dust node "sees" the beams of two lighthouses at the same time. We will discuss these issues in the next sections in order to lay the foundation for an implementation of the system.

4.2 A Realistic System

During the first experiments it turned out that actually building a lighthouse with a sufficiently exact parallel beam is very difficult, at least given the limited technical capabilities that were available to us. This has the unfortunate consequence, that the model described in Section 4.1 cannot directly be used due to the resulting high inaccuracies. To understand the reason of these inaccuracies, consider the following example, where we assume a beam width of 10cm. Even if the angle of beam spread is only 1° (instead of 0° for an ideal parallel beam), the width of the beam at a distance of 5m would be about 18.7cm, resulting in an error of almost 90%. The relative error could be reduced somewhat by increasing the width of the beam. However, a large beam width also results in a large and clumsy base station device.

Therefore, instead of building a system perfectly matching the requirements of Section 4.1, we have to adapt our model to a system which can actually be built. In order to develop such a model, we first have to examine ways of generating near-parallel beams.

4.2.1 Beam Generation

In order to keep the hardware and energy overhead on the Smart Dust nodes small, the beam must be easily detectable. Furthermore, the system should work with high accuracy even if the base station is far away (tens of meters, say) from the nodes. Therefore we decided to use a laser-based approach. As mentioned above, the beam should be as wide as possible in order to keep inaccuracies small. In order to achieve this, we use *two* lasers to create the *outline* of a parallel beam as depicted in the upper half of Figure 4. This makes no difference to a single wide beam, since we are only interested in the edges of the beam (i.e., change from "dark" to "light" and vice versa) in order to measure t_{beam} and t_{turn}.

Due to the narrow laser beams, the "virtual" parallel beam generated this way can only be seen from a single plane, however. In order to ensure that the beam can be seen from any point in the northern hemisphere of the lighthouse without defocusing the lasers, the laser beams have to scan this space in some way. The lower half of Figure 4 depicts two ways to achieve this. The first approach uses a small mirror mounted on a rotating axle under an angle of 45°. By pointing the laser at this mirror, the reflected rotating beam describes a plane. With commercial off the shelf technology we can easily achieve a rotation frequency of about 300Hz. The second approach uses a small deflectable MEMS mirror similar to the one used as part of the corner cube retroreflector (CCR). The MEMS mirror presented in [7], for example, operates at 35kHz and achieves a deflection angle of 25°. A laser beam pointed at such a mirror can thus sweep over an angle of 50° at a frequency of 35kHz.

Based on this approach, a lighthouse consists of a (slowly) rotating platform, on which two semiconductor laser modules and two rotating (or deflectable) mirrors are mounted. However, as mentioned at the beginning of Section 4.2, it is next to impossible to assemble all the pieces such that the resulting "virtual" wide beam is almost parallel. Therefore, we have to come up with a model which describes an imperfect but realistic system. The model discussed below is based on rotating mirrors, since we used this approach in our prototype implementation of the system. However, the model equally applies to a system based on deflectable mirrors.

4.2.2 The Lighthouse Model

We use Figure 5 to explain the lighthouse model. It shows a simplified top and side view of the lighthouse. Each view shows the two mirror's rotation axes and the corresponding reflected rotating laser beams. Note that in general the angle enclosed by the mirror rotation axis and the mirror will not be exactly 45° (i.e., $\beta_i \neq 0^\circ$) due to manufacturing limitations. Therefore, the rotating reflected laser beams will form two cones as depicted in Figure 5. Moreover, the two mirror's

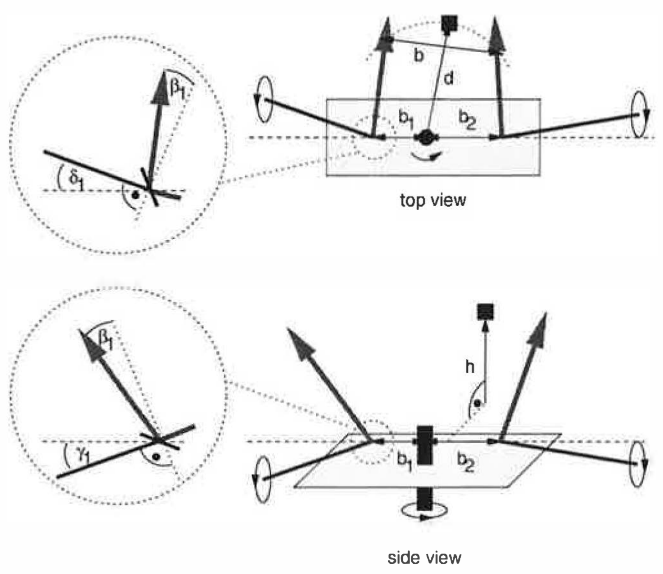


Figure 5: Model of a realistic lighthouse based on rotating mirrors. The zoom-ins show detail for one rotating mirror in the top and side views. The other rotating mirror has respective parameters β_2, γ_2, and δ_2.

rotation axes will not be perfectly aligned. Instead, the dashed vertical line (connecting the apexes of the two cones formed by the rotating laser beams) and the mirror rotation axes will enclose angles γ_i in the side view and angles δ_i in the top view that are different from 0°. Additionally, the figure shows the rotation axis of the lighthouse platform and its distances b_1 and b_2 to the apexes of the two light cones. The *lighthouse center* is defined as the intersection point of the lighthouse platform rotation axis and the dashed vertical line in Figure 5. Note that the idealistic lighthouse described in Section 4.1 is a special case of this more complex model with $\beta_i = \gamma_i = \delta_i = 0°$ and $b_1 = b_2$.

Now let us consider an observer (black square) located at distance d from the main lighthouse platform rotation axis and at height h over the lighthouse center. We are interested in the width b of the virtual wide beam as seen by the observer. Let us assume for this that we can build a lighthouse with $b_1 \approx b_2$ and $\beta_i, \gamma_i, \delta_i \approx 0°$, i.e., we do our best to approximate the perfect lighthouse described in Section 4.1. Then we can express b approximately as follows:

$$\begin{aligned} b \;\approx\;& b_1 + b_2 + \sqrt{d^2 + h^2}(\sin\beta_1 + \sin\beta_2) + \\ & h(\tan\gamma_1 + \tan\gamma_2) + d(\sin\delta_1 + \sin\delta_2) \end{aligned} \tag{6}$$

The inaccuracy results from the last two terms, which are linear approximations of rather complex non-linear expressions. For $\beta_1 = \beta_2 = 0°$, however, expression 6 becomes an equation. We will allow these factors to be built into the error term.

With $C^b := b_1 + b_2$, $C^\beta := \sin\beta_1 + \sin\beta_2$, $C^\gamma := \tan\gamma_1 + \tan\gamma_2$, and $C^\delta := \sin\delta_1 + \sin\delta_2$ we can rewrite expression 6 as

$$b \approx C^b + \sqrt{d^2 + h^2}C^\beta + hC^\gamma + dC^\delta \tag{7}$$

Note that C^b, C^β, C^γ, and C^δ are fixed lighthouse parameters. We will show below how they can be determined using a simple calibration procedure. We can express b also in terms of the angle α obtained using Equation 2:

$$b = 2d \sin\frac{\alpha}{2} \tag{8}$$

Combining expressions 7 and 8 we obtain the following expression which defines the possible (d, h) locations of the observer given a measured angle α and the lighthouse calibration values C^*:

$$2d \sin\frac{\alpha}{2} \approx C^b + \sqrt{d^2 + h^2}C^\beta + hC^\gamma + dC^\delta \tag{9}$$

Note that for given C^* and α the points in space whose d and h values are solutions of Equation 9 form a rotational hyperboloid centered at the rotation axis of the lighthouse. In the special case $\beta_i = \gamma_i = \delta_i = 0°$ and $b_1 = b_2$ this hyperboloid becomes a cylinder as in the idealistic model described in Section 4.1.

4.2.3 Location Computation

Similar to the idealistic model described in Section 4.1, the location of the observer can be obtained by determining the intersection point(s) of the three rotational hyperboloids defined by Equation 9. However, since the observed virtual beam width b now additionally depends on the height h of the observer, we have to take into account the exact lighthouse positions. Figure 6, which shows an extended version of Figure 3, illustrates this. The marks on the coordinate axes show the positions of the lighthouse center (as defined in Section 4.2.2) of each of the three lighthouses. That is, the coordinates of the observer are $(x_0 + h_x, y_0 + h_y, z_0 + h_z)$ with respect to the origin formed by the intersection of the three lighthouse rotation axes. In order to obtain approximations for the values h_x, h_y, and h_z, we have to solve the following equation system:

$$\begin{aligned} 2d_x \sin\tfrac{\alpha_x}{2} &= C_x^b + \sqrt{d_x^2 + h_x^2}C_x^\beta + h_xC_x^\gamma + d_xC_x^\delta \\ 2d_y \sin\tfrac{\alpha_y}{2} &= C_y^b + \sqrt{d_y^2 + h_y^2}C_y^\beta + h_yC_y^\gamma + d_yC_y^\delta \\ 2d_z \sin\tfrac{\alpha_z}{2} &= C_z^b + \sqrt{d_z^2 + h_z^2}C_z^\beta + h_zC_z^\gamma + d_zC_z^\delta \\ d_x^2 &= (y_0 + h_y)^2 + (z_0 + h_z)^2 \\ d_y^2 &= (x_0 + h_x)^2 + (z_0 + h_z)^2 \\ d_z^2 &= (x_0 + h_x)^2 + (y_0 + h_y)^2 \end{aligned} \tag{10}$$

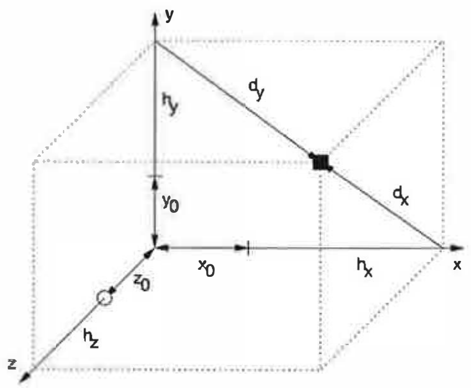


Figure 6: Positions of the lighthouses in the coordinate system. d_z is not shown for clarity.

The indices $\{x, y, z\}$ indicate which lighthouse the values are associated with. As with equation system 4, this system does not necessarily have a solution, since the parameters are only approximations obtained by measurements. Therefore, minimum mean square error (MMSE) methods have to be used to obtain approximations for the h_*. However, if the equation system 10 has a solution, we can approximately solve it by simple iteration. For this, we first transform each of the six equations of equation system 10 in order to obtain the following fixpoint form:

$$\begin{array}{rcl} h_x &=& f_1(d_x) \\ h_y &=& f_2(d_y) \\ h_z &=& f_3(d_z) \\ d_x &=& f_4(h_y, h_z) \\ d_y &=& f_5(h_x, h_z) \\ d_z &=& f_6(h_x, h_y) \end{array} \tag{11}$$

Note that we did not show arguments of the f_i (i.e., C_*^*, α_*, x_0, y_0, z_0) that do not change during iterative evaluation of the equation system. By using appropriate values for h_x^0, h_y^0, h_z^0, and Δ, we can obtain approximate solutions for h_x, h_y, h_z with the following algorithm:

```
$h_x := h_x^0$;
$h_y := h_y^0$;
$h_z := h_z^0$;
while (true) {
        $h'_x := f_1(f_4(h_y, h_z))$;
        $h'_y := f_2(f_5(h_x, h_z))$;
        $h'_z := f_3(f_6(h_x, h_y))$;
        if ($|h'_x - h_x| + |h'_y - h_y| + |h'_z - h_z| < \Delta$)
                break;
        $h_x := h'_x$;
        $h_y := h'_y$;
        $h_z := h'_z$;
}
```

At first, the h_* are initialized to the start values h_*^0. Using the f_i, new approximations h'_* are computed. We are finished if the new values are reasonably close to the original h_*. Otherwise we update the h_* to the new values and do another iteration. For good convergence of this algorithm the partial derivatives of the $f_i \circ f_{3+i}$ in the environment of the solution (h_x, h_y, h_z) should be small, which is typically true. In our prototype implementation we use $h_*^0 := 100\text{cm}$ and $\Delta := 0.1\text{cm}$. With this configuration, the algorithm typically performs 4-6 iterations.

4.2.4 Calibration

What remains to be shown is how we can obtain values for x_0, y_0, z_0, and C_x^*, C_y^*, C_z^*. Since the values x_0, y_0, z_0 are uncritical for the achieved accuracy, we assume they are measured directly. The C_*^* values, however, are very critical for the accuracy as was shown with the example at the beginning of Section 4.2. Therefore we have to perform a calibration.

For each of the three lighthouses we have to determine values for the four variables $C^b, C^\beta, C^\gamma, C^\delta$. For this, we place the observer at known locations (d_i, h_i) and obtain the respective α_i using equation 1. Doing so for at least four locations and using equation 9, we obtain the following linear equation system in $C^b, C^\beta, C^\gamma, C^\delta$:

$$\begin{array}{rcl} 2d_1 \sin\frac{\alpha_1}{2} &=& C^b + \sqrt{d_1^2 + h_1^2 C^\beta} + h_1 C^\gamma + d_1 C^\delta \\ 2d_2 \sin\frac{\alpha_2}{2} &=& C^b + \sqrt{d_2^2 + h_2^2 C^\beta} + h_2 C^\gamma + d_2 C^\delta \\ 2d_3 \sin\frac{\alpha_3}{2} &=& C^b + \sqrt{d_3^2 + h_3^2 C^\beta} + h_3 C^\gamma + d_3 C^\delta \\ 2d_4 \sin\frac{\alpha_4}{2} &=& C^b + \sqrt{d_4^2 + h_4^2 C^\beta} + h_4 C^\gamma + d_4 C^\delta \end{array} \tag{12}$$

As with the other equation systems, this system does not necessarily have a solution, since the parameters are only approximations obtained by measurements. Again, MMSE methods can be used to obtain approximations for the C^*. If the system has a solution, it can also be obtained by Gaussian elimination. For this, the d_i and h_i have to fulfill certain requirements. One simple rule of thumb is that both the d_i and the h_i should be pairwise distinct.

Note that calibration has to be performed only once for each base station (assuming that the system is stable enough and needs not be recalibrated) and is independent of the receiver nodes. Therefore, calibration can be performed using a more powerful receiver device than the limited Smart Dust node. As explained in Section 4.1, the base station broadcasts these calibration parameters to the Smart Dust nodes, which use them to compute their location using Equation System 10.

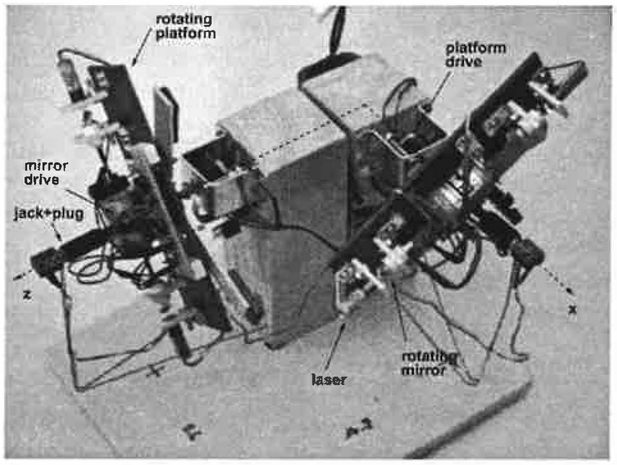


Figure 7: Prototype base station consisting of two lighthouses and the resulting 2D coordinate system.

4.3 Prototype Implementation

In order to evaluate the concepts developed in Section 4.2, we implemented a first prototype system. To keep the hardware overhead small, this prototype system consists of only two lighthouses and allows observers located on the plane $y = 0$ to determine their x and z coordinates. From a conceptual point of view, the differences to a 3D system are minimal.

4.3.1 The Base Station

Figure 7 shows a picture of the prototype base station. It consists of two mutually perpendicular lighthouses. The main lighthouse platform takes about $t_{\mathrm{turn}} = 60$s for one rotation. The platform is driven by a geared electro motor manufactured by FTB [34], which has a low flutter of about 0.1% of the rotation speed. Using an LM317 [36] adjustable voltage regulator, the voltage supply of the motor and thus the rotation speed of the platform can be adjusted. The two bars that extend from under the platform are used to move the center of gravity of the platform to the rotation axis, such that the platform rotates at a constant speed.

The power supply for the rotating platform is implemented by a stereo jack and associated plug. While the plug is fixed to the axle of the rotating platform, the jack is affixed to the chassis using a thin steel wire. This way, the round plug can rotate in the jack.

Beam generation is based on rotating mirrors as described in Section 4.2. Both rotating mirrors are driven by a single Graupner SPEED 280 electro-motor. In order to reduce vibrations, we did not use a rigid axle to connect the mirrors to the motor. Instead, we used small steel springs as axles. The rotating mirrors are supported by two ball bearings each. Two

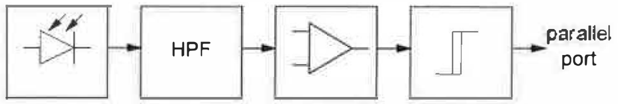


Figure 8: Schematic diagram of the receiver hardware.

1mW 650nm semiconductor laser modules with adjustable focus point their beam at the rotating mirrors.

The supply voltage of the motor and thus its rotation speed can be adjusted using an LM317 voltage regulator. The mirror rotation speeds of the two lighthouses are slightly different ($t_{\mathrm{mirror}} = 4$ms and $t_{\mathrm{mirror}} = 5$ms for one rotation, respectively), such that the observer can distinguish the two lighthouses based on the time interval between successive light flashes, which will be explained in more detail in Section 4.3.2. Hence, in order to detect a beam, the observer's photo detector must at least be hit twice by the rotating laser beam. Note that due to the fast rotation of the laser beams, the average light intensity is low enough to be eye-safe.

There is a slight chance that the photo detector is hit by the beams of both lighthouses at the same time. We will explain in Section 4.3.2 how an observer can detect and handle this situation. However, since the diameter of the laser beams is rather small, the likelihood of this event is small. By selecting slightly different platform rotation speeds for the two lighthouses, we can ensure that for each observer this happens only once in a while. In our experiments this happened about every 100 lighthouse rotations at a single fixed observer.

4.3.2 The Nodes

The receiver prototype consists of a small electronic circuit connected to the parallel port of a laptop computer running Linux. Figure 8 shows a schematic diagram of the receiver hardware. A photo diode converts the intensity of the incident light into a proportional voltage. The light that is incident to the photo diode mainly consists of three components:

- direct current (DC) components resulting from slowly changing daylight
- low frequency components resulting from artificial lighting powered with 50Hz alternating current (AC)
- higher frequency components resulting from laser light flashes at about 200Hz-300Hz ($1/t_{\mathrm{mirror}}$)

Since we are only interested in the higher frequency laser flashes, we run the output signal of the photo diode through a high pass filter (HPF) which removes DC and low frequency components. Due to this, the detector is insensitive to daylight and artificial light.

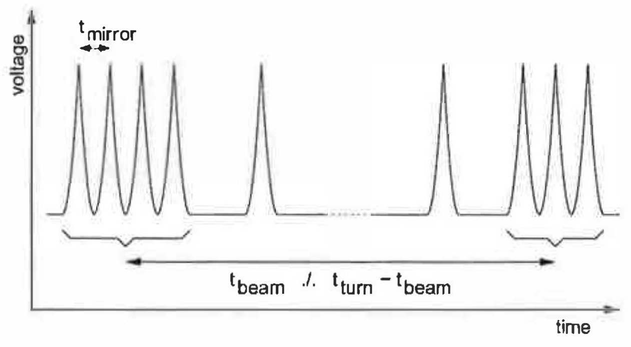


Figure 9: Input voltage at the parallel port as beams pass by the photo detector.

The output of the HPF is then amplified using an operational amplifier, whose output is in turn fed into a Schmitt Trigger. The latter implements a hysteresis, i.e., when the input voltage level exceeds a certain value V_1 it lowers the output voltage to a minimum. When the input voltage falls below a certain value V_2, the Schmitt Trigger raises the output voltage to a maximum. The output of the Schmitt Trigger is connected to the parallel port, so that each laser flash on the photo diode causes a parallel port interrupt to be triggered.

The receiver software consists of two main components, a Linux device driver which handles the parallel port interrupt, and an application level program which performs the actual location computation and lighthouse calibration. The device driver mainly consists of the parallel port interrupt handler, which is implemented using the parapin [37] parallel port programming library. Moreover, it implements a Linux special device `/proc/location`, which provides a simple interface to user level applications. By writing simple ASCII commands to this device, a user level program can instruct the device driver to do some action. By reading the `/proc/location` device, a user level program can obtain the current status and measured angle α according to Equation 1 of all detected lighthouses.

In order to measure α, the driver has to evaluate the interrupts it sees. To understand how this is done, consider Figure 9, which shows the input voltage at the parallel port over time. As the first rotating laser beam passes by the photo detector, the parallel port sees a sequence of sharp pulses resulting from the fast rotating mirror. The pulses stop if the lighthouse platform has turned enough so that the photo detector isn't hit any longer by the rotating beam. After some time, the second rotating beam passes by the photo detector and again generates a sequence of fast pulses.

Recall that each pulse generates an interrupt, which results in the device driver interrupt handler being invoked. The handler then uses the system clock (which has μs resolution under Linux) to determine the point in time when the interrupt occurred.

The time interval between two successive fast pulses equals the time t_{mirror} for one rotation of the mirror. Since each lighthouse has a different t_{mirror}, this value can be used to distinguish different lighthouses. Please note that the pulse sequences can contain "holes" where the laser beam missed the photo detector due to vibrations. The driver removes all peaks separated by holes from the beginning and the end of the sequence of pulses. The time median of the resulting shorter sequence of pulses without holes is assumed as the detection time of the beam (indicated by the braces in Figure 9).

Recall from Section 4.2, that we implemented a "virtual" wide beam by two rotating laser beams that form the outline of this wide beam. Therefore, the time passed between the medians of two successive packs is either t_{beam} or $t_{\mathrm{turn}} - t_{\mathrm{beam}}$. If the actual value is small (e.g., $< 1\,\mathrm{sec}$) then it is assumed to be t_{turn}. If the lighthouse has just been initialized the driver also measures $t_{\mathrm{turn}} - t_{\mathrm{beam}}$ in order to obtain t_{turn}. Since the latter does not change, this has to be done only once. Later on, the driver can output a new α with each round of the lighthouse.

In order to distinguish successive pulses from "holes", and holes from "beam switches", the driver knows tight lower and upper bounds for the possible values of t_{mirror} and t_{turn}. In Section 4.3.1 we mentioned the possibility that beams from different lighthouses may hit the photo receiver at the same time. If this happens the resulting time between successive pulse will fall below the lower bound for t_{mirror}, such that the driver can detect this situation instead of producing faulty results.

We also ported the receiver hardware and software to an ATMEL AT128L 8-bit embedded micro controller [38]. This setup more closely resembles the limited capabilities of a Smart Dust node and allows us to study the potential effects of a Smart Dust node on the location system.

4.4 Measurements

In this section we present an initial benchmark obtained with the prototype described above. We will begin by describing the calibration procedure.

4.4.1 Calibration

Calibration of the base station involves the following three steps:

- Ensuring that the lighthouses are mutually perpendicular.
- Measuring the offsets of the lighthouse centers x_0 and z_0.

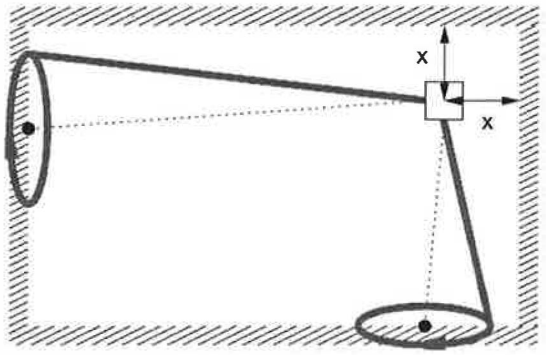


Figure 10: Ensuring mutually perpendicular lighthouses.

- Determining $C^b, C^\beta, C^\gamma, C^\delta$ for each of the two lighthouses.

In order to ensure that the two lighthouses are mutually perpendicular, we placed the base station at the corner of a rectangular room as depicted in Figure 10, such that the rotation axes of the two lighthouses are at distance x from the two perpendicular walls. We disabled the motors that drive the rotating mirrors and one of the two lasers of each lighthouse. Then we adjusted the mirror so that the remaining laser beam points at the opposite wall. Due to the rotating lighthouse platforms, the laser spots draw two circles on the walls. The centers of these two circles mark the position where the lighthouse rotation axes hit the wall as depicted in Figure 10. Now we adjust the lighthouses on the common chassis such that the centers of these circles also have a distance x from the walls. In our measurement setup, we placed the base station at x = 20cm in a room with a size of about 5m by 5m.

As mentioned in Section 4.2, the lighthouse center offsets x_0 and z_0 from the origin of the coordinate system defined by the lighthouse rotation axes are not critical for the accuracy of the system. Therefore, we measured them directly at the base station device.

In order to determine the C^* values, we placed the observer at the four locations $(x, z) \in \{(50,50),(480,80),(80,480),(340,340)\}$ (all values in centimeters) on the floor in the base station coordinate system. The respective lighthouse distance and height values are obtained from the (x, z) values as follows:

$$\begin{aligned} (d_x, h_x) &:= (z, x - x_0) \\ (d_z, h_z) &:= (x, z - z_0) \end{aligned} \tag{13}$$

At each of the locations we performed ten measurements of α for each lighthouse and computed the mean value. For each of the two lighthouses we then solved Equation System 12 in order to obtain the C^* values.

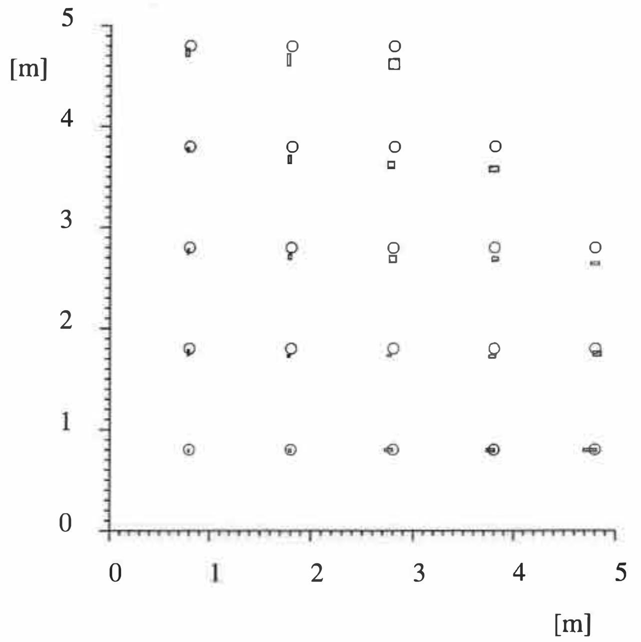


Figure 11: Location estimation benchmark. The ground truth locations are at the centers of the circles. The mean of the measured locations are at the centers of the boxes. The edge length of each box is twice the standard deviation in each axis.

4.4.2 Benchmark

For the benchmark, we placed the observer at 22 locations on the grid $(80cm + i * 100cm, 80cm + j * 100cm)$ in the base station coordinate system on the floor of the room. At each of the locations, we measured the location ten times by iteratively solving Equation System 10 as described in Section 4.2.

Figure 11 shows the base station coordinate system and the results of these measurements. Ground truth locations (x, z) are indicated by circles. The mean of the computed location $(\bar{x}, \bar{z})$ is at the center of the small boxes. The edge length of each box is twice the standard deviation s_x (s_z) of the measurements in the respective axis.

Please note that we determined ground truth locations using a cheap 5m tape measure, resulting in a maximum error of about $\pm$1cm in each axis. Also note that we did not perform out-lier rejection or any other statistical "tricks" to improve the mean values or standard deviations.

The mean relative offset of the mean locations from ground truth locations (i.e., $|\bar{x} - x|/x$) is 1.1% in the x axis, and 2.8% in the z axis. The overall mean relative offset of the mean locations from ground truth locations (i.e., $|(\bar{x}, \bar{z}) - (x, z)|/|(x, z)|$ is 2.2%. The mean relative standard deviation (i.e., s_x/x) is 0.71% in the x axis and 0.74% in the z axis. The overall mean relative standard deviation (i.e., $s_{|(x,z)|}/|(x, z)|$) is 0.68%.

Note that while the mean standard deviations are almost the same for both axes, the mean relative offset of 2.8% in the z axis is more than twice the value for the x axis. We believe that this is due to the way we performed calibration. Firstly, some of the locations where we performed measurements are outside of the convex hull of the locations where we performed calibration. Additionally, we calibrated at only four locations and solved Equation System 12 directly in order to obtain the C^* values. We expect better results by performing calibration using a larger set of reference locations and by using MMSE methods as mentioned in Section 4.2. We are currently working on improving the calibration part of our software.

4.5 System Analysis

This section presents a first analysis of several aspects of the Lighthouse location system, namely factors that influence the accuracy of the location estimates, limits on the maximum distance of nodes from the base station, the effects of node mobility, and the cost of adding location support to dust nodes.

4.5.1 Accuracy

In this section we want to examine which factors influence the accuracy of the system. For this, we have to examine errors that can occur during the measurement of t_{beam} and t_{turn}. From a measurement point of view the two are identical, since they are both an amount of time elapsed between two beam sightings. Therefore we will use t as a genus for the two and Δt as the absolute error of t. The following list contains possible causes for measurement errors:

- Vibrations: Due to their fast rotation, the mirrors and thus the reflected beams suffer from small vibrations, resulting in a small angle ϵ of beam spread, which is about 0.05° in our prototype. Assuming $\sin \epsilon = \epsilon$ (since $\epsilon \approx 0$), the resulting error is $\Delta t \leq t_{\text{turn}} \frac{\epsilon\sqrt{d^2+h^2}}{2\pi d}$ for an observer located at distance d from the lighthouse rotation axis and at height h over the lighthouse center.

- Lower bound on time t_{mirror} for one mirror rotation: Since we can measure elapsed time only when the rotating laser beam hits the photo detector, the accuracy of t_{beam} and t_{turn} is limited by the speed of the rotating mirrors (i.e., t_{mirror}). The resulting error is $\Delta t \leq t_{\text{mirror}}$.

- Flutter of platform rotation: The relative error in lighthouse rotation speed ρ_{lh} causes an error in t. ρ_{lh} is mainly caused by the flutter of the motor driving the lighthouse platform. The motor used in our prototype has a flutter of 0.1%. The resulting error is $\Delta t \leq t\rho_{\text{lh}}$.

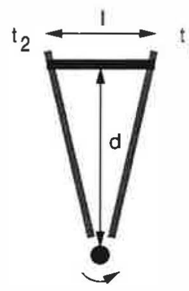


Figure 12: The photo detector must be hit by the laser beam at least twice.

- Variable delays: There is a variable time offset between the laser beam hitting the photo detector and the interrupt handler reading the clock. On the path from the photo detector to the interrupt handler are many sources of variable delay, such as hardware and interrupt latency. The actual value of this error pretty much depends on what is currently happening on the computer, but is typically small compared to the other sources of errors.

- Clock resolution: The minimum time unit t_{min} that can be measured by the clock limits the time resolution for measurement of t. The Linux laptop we used has $t_{\text{clockres}} = 1\mu\text{s}$. On the ATMEL we used a 16-bit counter to implement a clock with $t_{\text{clockres}} = 50\mu\text{s}$. The resulting error is $\Delta t \leq t_{\text{clockres}}$.

- Clock drift: The maximum relative error ρ_{clock} in the clock rate also causes an error in t. A typical value is $\rho_{\text{clock}} = 10^{-5}$ both on Linux and the ATMEL. The resulting error is $\Delta t \leq t\rho_{\text{clock}}$.

In our prototype systems, the clearly dominating errors are caused by vibrations, limited t_{mirror}, and flutter of platform rotation. The use of deflectable MEMS mirrors can both drastically reduce vibrations and t_{mirror}. The flutter of platform rotation can be reduced to about 0.01% by using electronically stabilized motors as used, for example, in turntable drives. By this, we expect a possible reduction of Δt by a factor of about 10.

Note, however, that the errors resulting from these three main sources can be modeled by a Gaussian noise source. This means that averaging over a large number of measurements helps to reduce the error.

4.5.2 Range

In this section we want to examine the maximum range, at which observers can still determine their location. This maximum range mainly depends on two issues.

The first of these issues is that the photo receiver has to be hit twice by each of the rotating beams in order for the receiver to identify the lighthouse as explained in Section 4.3.2. Figure 12 depicts this situation. It shows a top view of a lighthouse with only *one* of the two rotating beams at *two* points in time t_1 and t_2. At t_1, the beam hits the photo detector at distance d from the lighthouse rotation axis the first time. Then, the mirror does one rotation and hits the photo detector a second time at t_2. During $t_2 - t_1$, the lighthouse platform has rotated a bit to the left. l denotes the diameter of the photo detector. Assuming a constant diameter w of the laser beam, the distance d at which the photo detector is hit at least twice is given by the following inequality:

$$d < \frac{l + w}{2\sin(\pi t_{\text{mirror}}/t_{\text{turn}})} \tag{14}$$

With the values of our prototype system $l = 5\text{mm}$, $w = 3\text{mm}$, $t_{\text{mirror}} = 4\text{ms}$, $t_{\text{turn}} = 60\text{sec}$ we can achieve a theoretical maximum range of about 14m. This value can be improved by increasing t_{turn}, by decreasing t_{mirror}, or by defocusing the lasers a bit, such that there is a small angle of beam spread. However, there are certain limits for each of these possibilities. The angle of beam spread is limited by the sensitivity of the photo detector and the output power of the laser. t_{mirror} is limited by the possible maximum speed of the mirrors. With MEMS deflectable mirrors such as the one presented in [7], we can achieve $t_{\text{mirror}} = 1/35kHz = 30\mu s$. t_{turn} is limited by the frequency of location updates needed by the nodes and thus by the degree of node mobility (see Section 4.5.4).

The second issue that limits the maximum range of the system is the speed of the photo detector. Using COTS technology, the beam has to stay on the photo detector for about $t_{\text{photo}} = 10\text{ns}$ in order to be detected. Depending on the minimum retention period t_{photo} of the laser beam on the photo detector, the maximum distance d is limited according to the following inequality:

$$d < \frac{l + w}{2\sin(\pi t_{\text{photo}}/t_{\text{mirror}})} \tag{15}$$

With the current values of our prototype $t_{\text{photo}} = 200\text{ns}$, $t_{\text{mirror}} = 5\text{ms}$, $l = 4\text{mm}$, $w = 3\text{mm}$ we can achieve a theoretical maximum range of about 27m, giving us an overall range limit of 14m. Again, this value can be improved by reducing t_{mirror} and by defocusing the laser with the same limits as above.

The actually measured maximum range, at which the receiver prototype could still detect the base station is about 11 meters. However, the range can be increased by adjusting certain system parameters. A more elaborate system built using fast deflectable MEMS mirrors with values $l = 1\text{mm}$, $w = 20\text{mm}$ (due to beam spread), $t_{\text{mirror}} = 1\text{ms}$, $t_{\text{turn}} = 60\text{sec}$, and $t_{\text{photo}} = 10\text{ns}$, for example, could achieve a theoretical maximum range of about 210m (the minimum obtained from Inequalities 14 and 15). Based on our experience, we would expect a practical maximum range of about 120-140m of a system with these parameters, which approximately equals the maximum communication range of 150m during the day for the Berkeley experiments [19].

4.5.3 Cost

In this section we want to examine how the presented location system fits the stringent resource restrictions of future Smart Dust Systems. As explained in Section 3.3, these restrictions especially apply to the receiver side, i.e., the Smart Dust nodes.

The Berkeley Smart Dust prototype has already demonstrated that a photo detector similar to the one we are using for our location system is feasible. What remains to be shown is how the receiver software (i.e., the device driver and the user level program) fit onto a Smart Dust node.

Both the processing overhead and the memory footprint of the device driver are very low, which is very important for Smart Dust. The first is true because the driver is interrupt driven, i.e., it does not do any sensor sampling or polling. Moreover, the interrupt can be used to wake up the processor from a power saving mode. Thus, the system has to be woken up only during the short periods when a beam is hitting the photo detector. The memory footprint is very low because the driver does not have to store arrays of peak detections. Instead, for each sequence of peaks it only keeps "first peak" and "last peak" time stamps which are updated when a new interrupt occurs. The whole data structure for one lighthouse only takes about 25 bytes.

Similarly, the location computation part of the user level program has a very low memory footprint. It just retrieves the α values from the device driver and executes the approximation program described in Section 4.2. Given the relatively infrequent location updates, speed is not a problem. On computationally very limited platforms like future Smart Dust nodes, it might be necessary to revert to fixed point arithmetic and a hardware implementation of the location computation code in case the provided processing capabilities are too limited. Besides the basic arithmetic operations $(+, -, *, /)$ we need support for $\sin\alpha$ and $\sqrt{x}$ in order to solve Equation System 10. Note that $\sin\alpha$ is easy to approximate since the values of α obtained from Equation 1 are small due to $t_{\text{beam}} \ll t_{\text{turn}}$. The second order approximation $\sin\alpha \approx \alpha - \alpha^3/6$ has a maximum error of 0.1% for $|\alpha| \leq 33°$. There are also fast approximations for $y = \sqrt{x}$. One possible approach is to first approximate $1/\sqrt{x}$ by iterating $y := y(3 - xy^2)/2$ with an appropriate initial value for y. Multiplying the result by x gives an approximation for $\sqrt{x}$.

The requirements on the clock are also quite relaxed. Note that we don't need a real-time clock since we are only interested in the quotient $t_{\text{turn}}/t_{\text{beam}}$. A simple counter which ticks at a constant rate would also be sufficient. The resolution of the clock (or counter) just has to be high enough to reliably distinguish the t_{mirror} values of different lighthouses. Since the t_{mirror} values of our prototype system are 4ms and 5ms, respectively, a clock resolution of 0.5ms would be sufficient.

Please note that dust nodes don't have to be calibrated due to the following two reasons. Firstly, the two beam sightings used to measure t_{beam} and t_{turn} are identical from a measurement point of view. Any constant hardware and software delays will subtract out. Secondly, only the quotient $t_{\text{turn}}/t_{\text{beam}}$ is used for node localization, which is independent of the actual clock frequency.

4.5.4 Node Mobility

If nodes change their location over time, they have to update their location estimates frequently in order to avoid inaccuracies resulting from using outdated location estimates. Moreover, node movement during the measurement of parameters needed for location computation can cause inaccuracies in the estimated location.

The time t_{update} between successive location updates usually equals the time t_{turn} required for one rotation of the lighthouse. Thus, the update frequency $1/t_{\text{update}}$ can be increased by decreasing t_{turn}. However, there is an easy way to double the update frequency when using rotating mirrors for beam generation, because the beams are reflected to both sides of the lighthouse as depicted by the dashed laser beams in Figure 5. Thus, we actually have two "virtual" wide beams we can use for location estimation, effectively doubling the update frequency.

If a node moves during measurement of t_{beam} (i.e., after detection of the first beam and before detection of the second beam), the obtained value of t_{beam} will be incorrect. Additional errors are caused by the node moving between measurements of t_{beam} of the three lighthouses.

There are two ways to detect and reject faulty location estimates resulting from node movement during measurement. The first compares two or more consecutive position estimates and rejects them if they differ by more than a small threshold. The second approach uses accelerometers to detect movement during measurement. Accelerometers can also be used to estimate node movement (velocity, direction) during measurements of t_{beam}. The obtained values can be used to correct t_{beam}, such that correct location estimates can also be obtained during node movement. In fact, the Smart Dust prototypes developed at Berkeley already contain such sensors.

4.5.5 Line-Of-Sight Requirement

As mentioned in Section 2, communication between a node and the base station requires an uninterrupted line-of-sight (LOS) even for "plain" Smart Dust (i.e., without using the Lighthouse Location System). Hence, the presented location system does not introduce additional restriction with respect to LOS.

Temporary LOS obstructions can cause wrong position estimates if a dust node misses one of the laser beams. However, the probability of such errors can be reduced by comparing two or more consecutive positions estimates and rejecting them if they differ by more than a small threshold. Reflected laser beams are typically not detected by the receiver hardware, since diffuse reflection reduces the laser light intensity drastically.

Note that other localization systems based on ultrasound and radio waves provide location estimates even in the case of an obstructed line-of-sight. However, the resulting location estimates are typically wrong due to relying on signals reflected around the obstruction. Often it is difficult to detect such situations [14], which may result in using wrong location estimates unnoticed.

4.5.6 Robustness

We assume that base stations are immobile and mounted in a safe place (with respect to harmful environmental influences) due to their potential long range (see Section 4.5.2). On the other hand, dust nodes are subject to mobility and other kinds of environmental influences (e.g., LOS obstructions), which can cause faulty location estimates.

However, in Sections 4.5.4 and 4.5.5 we mentioned extensions to the basic system in order to detect and reject such faulty locations estimates with high probability. This leaves us in a situation, where dust nodes either obtain good position estimates or no at all.

5 Related Work

Research has developed numerous systems and technologies for automatically locating people, equipment, and other tangibles. [16] gives an excellent overview and taxonomy of such location systems. These systems all involve gathering data by sensing real-world physical quantities. The data is in turn used to compute a location estimate. Common systems use diffuse infrared light [26, 27, 29], visible light [8, 9, 33], laser light [22, 23], ultrasound [12, 13, 15, 24, 25, 30], and radio waves [2, 3, 4, 18].

Some systems have been specifically designed for use in multi-hop wireless ad hoc and sensor networks [3, 4, 6, 10, 13, 25] and do not require any external hardware infrastructure besides the nodes of the network itself. Other systems rely on an external infrastructure typically consisting of many devices, which have to be carefully placed in the environment of the objects being located [2, 15, 24, 29, 30].

However, the special characteristics of future Smart Dust systems as described in Section 2 and the resulting requirements for a location system as described in Section 3 rule out the usage of all of these location systems. The small size and limited resources rule out systems based on radio waves and ultrasound, since transducers for these physical media are too large and transceivers consume too much energy for Smart Dust nodes [19]. The employed optical single-hop communication rules out all systems which require neighbor-to-neighbor or multi-hop communication. The use of a single (or few) base station(s) rules out all system which require substantial external infrastructure. The required localized location computation (see Section 3.2) rules out all systems where nodes cannot compute their location on their own. Additionally, many systems (e.g., ones based on vision and broadband ultrasound) typically have a high processing overhead and large memory footprint due to the necessary signal processing on the raw input data (e.g., time series of images or audio samples).

The systems that come closest to fulfilling the requirements of Smart Dust are ones based on vision or laser ranging techniques. Laser ranging systems are based on measuring the distance between the laser ranger device and some passive object by a variety of different methods [35]. However, with all these methods, only the active laser ranger can estimate the distance, not the object being located, which precludes localized location computation. The same is true for vision based methods, where a high resolution video camera is used to estimate node location [8]. There are also systems which combine laser ranging and vision-based methods [23], which obviously suffer from the same problem.

6 Conclusion

Future Smart Dust systems present a novel set of challenges to a location system. We examined these challenges and found that location systems developed in the past for mobile computing systems and COTS sensor networks are not applicable to Smart Dust systems due to the novel characteristics of the latter.

We presented the Lighthouse location system for future Smart Dust systems. By extending the base station, this system allows Smart Dust to autonomously estimate their physical location with respect to the base station with high precision over distances of tens of meters without node calibration. Besides the single modified base station, the system does not require any additional infrastructure components. This is achieved by a new cylindrical lateration method. In contrast to traditional spherical methods, this approach does not have a wide baseline requirement. On the receiver side, only a simple optical receiver (amplified photo diode), moderate processing capabilities, and little memory are needed. That is, only marginal changes to the Smart Dust prototype developed at UC Berkeley are required.

We presented a prototype implementation of the system, a set of initial measurements, and a first analysis of several aspects of the system. Currently we are working on better support for node mobility and MMSE-based calibration. We are also currently analysing the system in more detail.

We plan to build a second revision of the base station prototype based on deflectable MEMS mirrors, which is expected to feature much improved accuracy, size, and power consumption.

Future work also includes an analysis of how a real Smart Dust implementation influences the quality of the location estimates. This includes factors like reduced clock resolution, increased clock skew, and the approximations mentioned in Section 4.5.3.

7 Acknowledgements

Many thanks to Tim Kindberg, Friedemann Mattern, Harald Vogt, and the MobiSys reviewers for valuable comments that helped improve this work.

References

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless Sensor Networks: A Survey. *Computer Networks*, 38(4):393--422, March 2002.

[2] P. Bahl and V. N. Padmanabhan. RADAR: An In-Building RF-based User Location and Tracking System. In *Infocom 2000*, Tel-Aviv, Israel, March 2000.

[3] J. Beutel. Geolocation in a PicoRadio Environment. Master's thesis, UC Berkeley, 1999.

[4] N. Bulusu, J. Heideman, and D. Estrin. GPS-less Low Cost Outdoor Localization for Very Small Devices. *IEEE Personal Communications*, 7(5):28–34, October 2000.

[5] N. Bulusu, J. Heideman, and D. Estrin. Adaptive Beacon Placement. In *ICDCS 2001*, Phoenix, USA, April 2001.

[6] S. Capkun, M. Hamdi, and J. P. Hubaux. GPS-free Positioning in Mobile Ad Hoc Networks. In *34th International Conference on System Sciences*, Hawaii, January 2001.

[7] R. Conant, J. Nee, K. Lau, and R. Muller. A Fast Flat Scanning Micromirror. In *2000 Solid-State Sensor and Actuator Workshop*, Hilton Head, USA, June 2000.

[8] T. Darrell, G. Gordon, M. Harville, and J. Woodfill. Integrated Person Tracking Using Stereo, Color, and Pattern Recognition. In *Conference on Computer Vision and Pattern Recognition*, Santa Barbara, USA, June 1998.

[9] D. L. de Ipina. Video-based Sensing for Wide Deployment of Sentient Spaces. In *Workshop on Ubiquitous Computing and Communications*, Barcelona, Spain, September 2001.

[10] L. Doherty, K. S. J. Pister, and L. E. Ghaoui. Convex Position Estimation in Wireless Sensor Networks. In *Infocom 2001*, Anchorage, Alaska, April 2001.

[11] D. Estrin, D. Culler, K. Pister, and G. Sukhatme. Connecting the Physical World with Pervasive Networks. *IEEE Pervasive Computing*, 1(1):59–69, January 2002.

[12] E. Foxlin, M. Harrington, and G. Pfeifer. A Wide-Range Wireless Motion-Tracking System for Augmented Reality and Virtual Set Applications. In *25th Annual Conference on Computer Graphics*, Orlando, USA, July 1998.

[13] L. Girod, V. Bychkovskiy, J. Elson, and D. Estrin. Locating Tiny Sensors in Time and Space: A Case Study. In *International Conference on Computer Design (ICCD) 2002*, Freiburg, Germany, September 2002.

[14] L. Girod and D. Estrin. Robust range estimation using acoustic and multimodal sensing. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS) 2001*, Maui, Hawaii, October 2001.

[15] M. Hazas and A. Ward. A Novel Broadband Ultrasonic Location System. In *Ubicomp 2002*, Gothenburg, Sweden, September 2002.

[16] J. Hightower and G. Borriello. Location Systems for Ubiquitous Computing. *IEEE Computer*, 34(8):57–66, August 2001.

[17] J. Hightower, C. Vakili, G. Borriello, and R. Want. Design and Calibration of the SpotON Ad-Hoc Location Sensing System. unpublished, August 2001.

[18] B. Hofmann-Wellenhof, H. Lichtenegger, and J. Collins. *Global Positioning System: Theory and Practice, 4th Edition.* Springer-Verlag, 1997.

[19] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Emerging Challenges: Mobile Networking for Smart Dust. *Journal of Communications and Networks*, 2(3):188–196, September 2000.

[20] M. Langheinrich. Privacy by Design – Principles of Privacy-Aware Ubiquitous Systems. In *International Conference on Ubiquitous Computing (Ubicomp 01)*, Atlanta, GA, September 2001.

[21] M. Last and K. S. J. Pister. 2-DOF Actuated Micromirror Designed for Large DC Deflection. In *3rd Intl. Conf. on Micro Opto Electro Mechanical Systems (MEOMS 99)*, Mainz, Germany, August 1999.

[22] U. Lohr. Digital Elevation Models by Laserscanning: Principle and Applications. In *3rd Intl. Airborne Remote Sensing Conference*, Copenhagen, Denmark, 1997.

[23] B. Nickerson, P. Jasiobedzki, D. Wilkes, M. Jenkin, E. Milios, J. Tsotsos, A. Jepson, and O. N. Bains. The ARK Project: Autonomous Mobile Robots for Known Industrial Environments. *Robotics and Autonomous Systems*, 1(25):83–104, 1998.

[24] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The Cricket Location-Support System. In *Mobicom 2000*, Boston, USA, August 2000.

[25] A. Savvides, C. C. Han, and M. Srivastava. Dynamic Fine-Grained Localization in Ad-Hoc Networks of Sensors. In *Mobicom 2001*, Rome, Italy, July 2001.

[26] B. N. Schilit, N. Adams, R. Gold, M. Tso, and R. Want. The ParcTab Mobile Computing System. In *4th Workshop on Workstation Operating Systems*, Napa, USA, October 1993.

[27] T. Starner, D. Kirsh, and S. Assefa. The Locust Swarm: An Environmentally Powered, Networkless Location and Messaging System. In *ISWC 1997*, Cambridge, USA, October 1997.

[28] H. Wang, D. Estrin, and L. Girod. Preprocessing in a Tiered Sensor Network for Habitat Monitoring. Submitted for publication, September 2002.

[29] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems*, 10(1):91–102, 1992.

[30] A. Ward, A. Jones, and A. Hopper. A New Location Technique for the Active Office. *IEEE Personal Communications*, 4(5):42–47, October 1997.

[31] B. Warneke, M. Last, B. Leibowitz, and K. S. J. Pister. Smart Dust: Communicating with a Cubic-Millimeter Computer. *IEEE Computer Magazine*, 34(1):44–51, January 2001.

[32] K. Whitehouse and D. Culler. Calibration as Parameter Estimation in Sensor Networks. In *Workshop on Wireless Sensor Networks and Applications (WSNA) 02*, Atlanta, USA, September 2002.

[33] C. Wren, A. Azarbayejani, T. Darrell, and A. Pentland. Pfinder: Realtime Tracking of the Human Body. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(7):780–785, July 1997.

[34] FTB Feintechnik Bertsch. www.ftb-bertsch.de.

[35] Laser Distance Measurements. cord.org/cm/leot/Module6/module6.htm.

[36] National Semiconductors. www.national.com.

[37] parapin - a Parallel Port Programming Library for Linux. www.circlemud.org/~jelson/software/parapin/.

[38] Smart-Its Project. www.smart-its.org.

Anonymous Usage of Location-Based Services Through Spatial and Temporal Cloaking

Marco Gruteser and Dirk Grunwald
Department of Computer Science
University of Colorado at Boulder
Boulder, CO 80309
{gruteser,grunwald}@cs.colorado.edu

Abstract

Advances in sensing and tracking technology enable location-based applications but they also create significant privacy risks. *Anonymity* can provide a high degree of privacy, save service users from dealing with service providers' privacy policies, and reduce the service providers' requirements for safeguarding private information. However, guaranteeing anonymous usage of location-based services requires that the precise location information transmitted by a user cannot be easily used to re-identify the subject. This paper presents a middleware architecture and algorithms that can be used by a centralized location broker service. The adaptive algorithms adjust the resolution of location information along spatial or temporal dimensions to meet specified anonymity constraints based on the entities who *may* be using location services within a given area. Using a model based on automotive traffic counts and cartographic material, we estimate the realistically expected spatial resolution for different anonymity constraints. The median resolution generated by our algorithms is 125 meters. Thus, anonymous location-based requests for urban areas would have the same accuracy currently needed for E-911 services; this would provide sufficient resolution for wayfinding, automated bus routing services and similar location-dependent services.

1 Introduction

Improvements in sensor and wireless communication technology enable accurate, automated determination and dissemination of a user's or object's position [1, 2]. There is an immense interest in exploiting this positional data through location-based services (LBS) [3, 4, 5, 6]. For instance, LBSs could tailor their functionality to the user's current location, or vehicle movement data would improve traffic forecasting and road planning.

However, without safeguards, extensive deployment of these technologies endangers users' location privacy and exhibits significant potential for abuse [7, 8, 9]. Common privacy principles demand, among others, user consent, purpose binding,[1] and adequate data protection for collection and usage of personal information [10]. Complying with these principles generally requires notifying users (data subjects) about the data collection and the purpose through privacy policies; it also entails implementing security measures to ensure that collected data is only accessed for the agreed-upon purpose.

This paper investigates a complimentary approach that concentrates on the principle of minimal collection. In this approach location-based services collect and use only de-personalized data—that is, *practically anonymous* data [11]. This approach promises benefits for both parties. For the service provider, practically anonymous data causes less overhead. It can be collected, processed, and distributed to third parties without user consent. For data subjects, it removes the need to evaluate potentially complex service provider privacy policies.

Practical anonymity requires that the subject cannot be reidentified (with reasonable efforts) from the location data. Consider a message to a road map service that comprises a network address, a user ID, and coordinates of the current location. Identifiers like the user ID and the network address are obvious candidates for reidentification attempts. For anonymous service usage, the user ID can be omitted and the network address problem is addressed by mechanisms such as Crowds [12] or Onion Routing [13], which provide sender anonymity.

However, revealing accurate positional information can pose even more serious problems. Consider a bus wayfinding application that overlays bus route and arrival information, such as that marketed by NextBus [14]. The Global Positioning System (GPS) typically provides 10–30 foot accuracy, and this accuracy can be increased using enhancement techniques, such as differential GPS. A location-based service could query a bus transit server and return information about buses in the current vicinity and when they will arrive at various stops. By issuing such a query, the location-based service has learned information about the application user, including her location and some network identity information. This location information can be correlated with public knowledge to reidentify a user or vehicle. For example, when the service is used while still parked in the garage or on the driveway, the location coordinates can be mapped to the address and the owner of the residence. If queries

[1] When seeking user consent, data collectors need to explain the specific purpose for which the data will be used. Subsequent use for other purposes is prohibited without additional user approval.

are sufficiently frequent, they can be used to track an individual. Note that this tracking uses publicly available information as opposed to the identity behind network addresses. The privacy problems are magnified if location information is recorded and distributed continuously as envisioned in telematics applications such as "pay as you drive" insurance, traffic monitoring, or fleet management. In this case an adversary not only learns about network services that a subject uses but also can track the subjects movements and thus receives real-world information such as frequent visits to a medical doctor, nightclub, or political organizations.

Anonymity in LBSs must be addressed at multiple levels in the network stack depending on what entities can be trusted. This paper approaches the problem of anonymity at the application layer by giving service providers access to anonymous location information; that is, information that is sufficiently altered to prevent re-identification. It contributes the following key ideas:

- a formal metric for location anonymity
- an adaptive quadtree-based algorithm that decreases the spatial resolution of location information to meet a specified anonymity constraint
- an algorithm that yields higher spatial resolution through decreasing temporal resolution for the same anonymity constraint
- an evaluation of the expected resolution for these algorithms based on traffic models comprised of cartographic material and automotive traffic counts

The structure of the paper is as follows: First we review related work in the areas of location privacy, anonymous communication, and privacy-aware databases. In Section 3 we describe location-based service scenarios from the telematics domain and discuss their data accuracy requirements. Section 4 then analyzes privacy threats caused by the location information used in LBSs. We continue by developing the concept of k-anonymous location information and an algorithm for cloaking too precise information in section 5. After that, we describe our implementation and evaluation based on automotive traffic models and present the corresponding results. Finally, we discuss the usefulness of the cloaking algorithms as well as security and anonymity properties of the system.

2 Related Work

Prior work on privacy aspects of telematics and location-based applications has mostly focused on a policy-based approach [15, 16]. Data subjects need to evaluate and choose privacy policies offered by the service provider. These policies serve as a contractual agreement about which data can be collected, for what purpose the data can be used, and how it can be distributed. Typically, the data subject has to trust the service provider that private data is adequately protected. In contrast, the anonymity-based approach de-personalizes data before collection, thus detailed privacy-policies and safeguards for data are not critical.

Specifically, the IETF Geopriv working group [15] is addressing privacy and security issues regarding the transfer of high resolution location information to external services and the storage at location servers. It concentrates on the design of protocols and APIs that enable devices to communicate their location in a confidential and integrity-preserving manner to a location server. The location server can reduce the data's resolution or transform it to different data formats, which can be accessed by external services if the data subject's privacy policy permits. The working group is also interested in enabling unidentified or pseudonymous transfer of location information to the server and access from the server. However, it does not claim that this provides a sufficient degree of anonymity.

The Mist routing project for mobile users [17] combines location privacy with communication aspects. It addresses the problem of routing messages to a subject's location while keeping the location private from the routers and the sender. To this end, the system comprises a set of mist routers organized in a hierarchical structure. The leaf nodes have knowledge of user locations but not their identities. They refer to them through handles (or pseudonyms). Each user selects a higher-level node in the tree, which acts as a semi-trusted proxy. It knows the identity of the user but not his exact location. The paper then presents a cryptographic protocol to establish connections between users and their semi-trusted proxies and mechanisms to connect to communication partners through their proxies. The paper does not address the problem of sending anonymous messages to external location-based services.

Location privacy has also been studied in position sensor systems systems. The Cricket system [1] places location sensors on the mobile device as opposed to the building infrastructure. Thus, location information is not disclosed during the position determination process and the data subject can choose the parties to which the information should be transmitted. Smailagic and Kogan describe a similar approach for a wireless LAN based location system [18]. However, these solutions do not provide for anonymity when location information is intentionally revealed.

Anonymous communication in packet-switching networks and web browsing has received a fair amount of attention. The fundamental concept of a mix has been proposed by Chaum [19] for email communications that are untraceable even for eavesdroppers and intermediary routers. A mix is a message router that forwards messages with the objective that an adversary cannot match incoming messages to outgoing messages. In particular, such Chaum-mixes have the following properties: messages are padded to equal size, incoming and outgoing messages are encrypted with different keys, messages are batched and reordered, and replay of incoming messages is prevented. Pfitzmann and colleagues [20] extend this mechanism to communication channels with continuous, delay-sensitive voice traffic.

Onion Routing [21] implements this anonymization protocol for an IP network layer and is applicable to both connection-based and connectionless protocols. In an initialization phase, the sender determines a route through a series of onion routers. The sender then repeatedly adds routing information to the payload and encrypts it using the onion routers public key. The result is an onion consisting of several layers of encryption that are stripped off while the packet passes through the router. Since the onion routers act as mix routers, it is difficult to trace the path of a data packet through the network.

Crowds [12] adapts a rerouting system for anonymous web browsing. This system focuses on protecting against individual adversaries, such as the web server, or a number of compromised routers. It does not require encryption techniques, because it relies on the jondos (mix routers) to be set up in different administrative domains. Thus no party has a global network view over all jondos. The Anonymizer service [22] has a similar goal, whereby users need to trust the single service provider. Finally, Hordes [23] reduced the performance overhead inherent in such rerouting systems by exploiting multicast communications and Guan *et al.* [24] contributed an analysis of anonymity properties of these systems using the probabilistic method.

In the database community, a large amount of literature exists on security control in statistical databases, which is covered by Adam and Wortmann's survey [25]. This research addresses the problem wherein a database should grant access to compute statistical functions (sum, count, average, etc.) on the data records only under the condition that the results do not reveal any specific data record. Approaches fall into the categories conceptual, input data perturbation, query restriction, and output perturbation; the solution we propose in this paper is similar to input data perturbation.

Instead of statistical point estimates, Agrawal and Srikant [26] describe how to obtain estimates of the distribution of values in confidential fields, which are suitable for data-mining algorithms. Confidential values are perturbed by adding a uniformly distributed random variable. The distribution of the original values can then be estimated through a Bayesian reconstruction procedure. An improved reconstruction procedure is described in [27].

Samarati and Sweeney [28] have developed generalization and suppression techniques for values of database tables that safeguard the anonymity of individuals. While this research is similar in goal, our work differs in that we protect dynamic data delivered from sensors as opposed to static database tables.

3 Accuracy Requirements of Location-Based Telematics Services

A key question for developing an anonymous LBS is: How accurate does a location based service need to be in order to provide useful information? It proves difficult to determine minimum accuracy requirements, since, from the service provider's perspective, more accurate information is generally more useful. However, we attempt to convince the reader that more general information is still sufficient for a large class of services by reviewing example services and the E-911 requirements on mobile phone carriers.

In October 1996, the United States Federal Communications Commission mandated the implementation of position systems for wireless 911 emergency callers (E-911) [29]. This service is designed to provide emergency rescue and response teams with the location of a cell phone emergency call, comparable to the traditional "911" service for regular phones. In the final phase, wireless carriers are required to estimate the caller's position with an accuracy of 125 m (RMS) in 67 percent of cases. The details have subsequently been subject to debate, but this initial requirement gives an indication of the expected accuracy. The location systems developed for the E-911 requirement have been widely regarded as an enabling technology for location-based services; therefore, we will regard this level of accuracy as useful.

3.1 System Assumptions

We assume that clients communicate position information to a location server with very high precision; in other words, the network client actually provides an accurate location to the location server. Position determination can be implemented either on the client itself (e.g., GPS) or by the wireless service provider, for example through triangulation of the wireless signal (hybrid approaches are also possible). To our knowledge, mobile phone operators in the United States found it challenging to meet the E-911 accuracy requirements through the latter approach. Thus, GPS information is likely far more accurate and privacy sensitive. Location-based service providers access location information through the location server. The full system comprises a location information source, a wireless network, location servers, and LBS servers. In a typical system, location information is determined by a location information source such as a GPS receiver in a vehicle. It is then periodically transmitted through a cellular or wireless network to the location server. When a vehicle sends a message or request to an LBS, the service accesses the vehicle's current location information from the location server, which acts as a proxy or middleware agent.

Finally, this paper focuses on services that do not require the user to log in and or present any kind of identifying information at the application layer. We believe that such LBSs will become available analogous to free services over the Internet. However, it would be interesting to extend this research to pseudonymous LBSs, which would allow tailoring services to individual users, for example.

3.2 Scenarios

To illustrate different accuracy requirements of location-based services, we provide three typical automotive telematics scenarios: Driving Conditions Monitoring, Road Hazard Detection, and a Road Map. Services are differentiated along the following dimensions:

- Frequency of Access
- Time-accuracy / Delay sensitivity
- Position accuracy

Table 1 presents a summary of the resulting requirements.

Driving Conditions Monitoring

Modern vehicles carry a variety of sensors that can determine weather and road conditions. Instead of deploying an expensive array of fixed sensors alongside highways, highway operators could obtain this information from the in-vehicle sensors. For example, the rain sensor built into high-end windshield wipers detects rainfall; additionally, traction control systems can report slippery or icy road conditions. The operator might respond to this information by dynamically adjusting speed limits on the highway.

Weather phenomena and corresponding road conditions typically cover larger areas. In addition, most warnings and speed limits must be given well ahead of the hazardous conditions. Thus, highly accurate position information is not necessary; about 100m road segments should be a suitable resolution for most cases. Conditions also do not change very abruptly, thus updates with a few minutes delay can be tolerated. In order to detect a change in driving conditions the external application needs quasi-continuous access to location information.

Road Hazard Detection

Dangerous, near-accident situations could be inferred from braking or electronic stability systems. Additionally, crash sensors for airbag deployment detect severe accidents. This information could be exploited to automatically generate statistics about the accident risk at intersections and road segments. These statistics are valuable for deciding on accident prevention measures.

Since this application collects longer-term statistics, information delay is not important and time accuracy requirements are low. For example, it would be useful to distinguish night and daylight situations or rush hour from mid-day traffic but not to collect information with second-resolution. Precise location information is crucial, however, to pinpoint dangerous spots such as intersections or pedestrian crossings.

Road Map

Drivers might request information related to their current location from LBSs. For example, the driver can ask for an area map or nearby hotels. The current location can be automatically obtained from the GPS sensor of a vehicle navigation system.

Response times of these services are important, thus, this application requires high time accuracy. The location, however, can be transmitted with medium accuracy; about 100m accuracy should be sufficient for obtaining point-of-interest information and area maps. Location is revealed only sporadically, when the driver issues requests. If such systems are used for navigation, the location can be revealed much more frequently.

4 Privacy Threats Through Location Information

We assume that an adversary seeking to violate anonymity may be able to intercept wireless and wired communications, may obtain data from the service provider's systems, and may have prior knowledge about a subject, whose messages he seeks to identify.

Our main concern is to prevent an accumulation of identifiable location information in service providers systems. LBS providers, without any malicious intent, will likely log service requests, similar to a web server that logs requested URLs and source IP addresses of the requester. Logs that include location information would open the door for subpoenas in court (e.g., divorce) proceedings, or individual adversaries who obtain a subject's location information under a pretext. Moreover, a less conscientious service provider might seek to identify subjects for marketing purposes or sell location records to third parties. In these cases, an adversary targets a large number of subjects, or seeks to obtain a location history for a particular subject from the records of a service provider.

A different type of adversary seeks to track future movements of a particular subject. However, such location information can also be obtained through traditional investigative methods such as shadowing a subject or mounting a location transmitter to a vehicle. These methods are related to the LBS problem in that they define a currently accepted level of protection. We consider the protection of anonymous LBSs sufficient if location tracking requires effort comparable to the traditional methods.

4.1 Threats

We distinguish two classes of privacy threats related to location-based services: communication privacy threats and location privacy threats. In the communication privacy domain, this paper concentrates on sender anonymity, meaning that eavesdroppers on the network and LBS providers cannot determine the originator of a message. Compared to non-LBS web services, the location information is the key problem: an adversary can

Service	Position Accuracy	Time Accuracy	Frequency of Access
Driving Conditions Monitoring	100 meters	minutes	continuous
Road Hazard Detection	10 meters	$>$ days	sporadic
Road Map	100 meters	sub-second	sporadic

Table 1: Approximate accuracy requirements of telematics services

reidentify the sender of an otherwise anonymous message by correlating the location information with prior knowledge or observations about a subject's location.

Consider the case where a subject reveals her location L in a message M to a location-based service and an adversary A has access to this information. Then, sender anonymity and location privacy is threatened by location information in the following ways:

Restricted Space Identification. If A knows that space L exclusively belongs to subject S then A learns that S is in L and S has sent M. For example, when the owner of a suburban house sends a message from his garage or driveway, the coordinates can be correlated with a database of geocoded postal addresses (e.g., [30]) to identify the residence. An address lookup in phone or property listings then reveals the owner and likely originator of the message.

Observation Identification. If A has observed the current location L of subject S and finds a message M from L then A learns that S has sent M. For example, the subject has revealed its identity and location in a previous message and then wants to send an anonymous message. The later message can be linked to the previous one through the location information.

Location Tracking. If A has identified subject S at location L_i and can link series of location updates $L_1, L_2, \dots, L_i, \dots, L_n$ to the subject, then A learns that S visited all locations in the series.

Location privacy threats describe the risk that an adversary learns the locations that a subject visited (and corresponding times). Through these locations, the adversary receives clues about private information such as political affiliations, alternative lifestyles, or medical problems. Assuming that a subject does not disclose her identity at such a private location, an adversary could still gain this information through location tracking. If the subject transmits her location with high frequency, the adversary can, at least in less populated areas, link subsequent location updates to the same subject. If at any point the subject is identified, her complete movements are also known.

5 Anonymizing Location Information

In our system model, the mobile nodes communicate with external services through a central anonymity server that is part of the trusted computing base. In an initialization phase, the nodes will set up an authenticated and encrypted connection with the anonymity server. When a mobile node sends position and time information to an external service, the anonymity server decrypts the message, removes any identifiers such as network addresses, and perturbs the position data according to the following cloaking algorithms to reduce the reidentification risk. Moreover, the anonymity server acts as a mix-router [19], which randomly reorders messages from several mobile nodes, to prevent an adversary from linking ingoing and outgoing messages at the anonymity server. Finally, the anonymity server forwards the message to the external service.

For designing the perturbation algorithms, we start with the assumption that the anonymity server knows the current position of all subjects. The subject's mobile nodes could periodically update their position information with the anonymizer.

5.1 k-Anonymous Location Information

While *anonymity* is etymologically defined as "being nameless" or "of unknown authorship" [31], information privacy researchers interpret it in a stronger sense. According to Pfitzmann and Koehntopp, "anonymity is the state of being not identifiable within a set of subjects, the anonymity set"[11]. Inspired by Samarati and Sweeney [28], we consider a subject as *k-anonymous* with respect to location information, if and only if the location information presented is indistinguishable from the location information of at least k 1 other subjects.

Unless otherwise stated, we assume that location information includes temporal information (i.e., when the subject was present at the location). More specifically, location information is represented by a tuple containing three intervals $([x_1, x_2], [y_1, y_2], [t_1, t_2])$. The intervals $[x_1, x_2]$ and $[y_1, y_2]$ describe a two dimensional area where the subject is located. $[t_1, t_2]$ describes a time period during which the subject was present in the area. Note that the intervals represent uncertainty ranges; we only know that at some point in time within the temporal interval the subject was present at some point of the area given by the spatial intervals. Thus, a location tuple for a subject is k-anonymous, when it describes not only the location of the subject, but also the locations of k 1 other subjects. In other words, k 1 other subjects also must have been present in the area and the time period described by the tuple. Generally speaking, the larger the anonymity set k is, the higher is the degree of anonymity. Thus, we will measure the degree of anonymity as the size of the anonymity set.

5.2 Adaptive-Interval Cloaking Algorithms

The key idea underlying this algorithm is that a given degree of anonymity can be maintained in any location—regardless of population density—by decreasing the accuracy of the revealed spatial data. To this end, the algorithm chooses a sufficiently large area, so that enough other subjects inhabit the area to satisfy the anonymity constraint.

The desired degree of anonymity is specified by the parameter k_{min}, the minimum acceptable anonymity set size. Furthermore, the algorithm takes as inputs the current position of the requester, the coordinates of the area covered by the anonymity server, and the current positions of all other vehicles/subjects in the area.

The spatial discretization algorithm that identifies a sufficiently large area for a given k_{min} is described in more detail in Table 2. In summary, the algorithm is inspired by quadtree algorithms [32]. It subdivides the area around the subject's position until the number of subjects in the area falls below the constraint k_{min}. The previous quadrant, which still meets the constraint, is then returned.

An orthogonal approach to spatial cloaking is temporal cloaking. This method can reveal spatial coordinates with more accuracy, while reducing the accuracy in time. The key idea is to delay the request until k_{min} vehicles have visited the area chosen for the requestor. The spatial cloaking algorithm is modified to take an additional spatial resolution parameter as input. It then determines the monitoring area by dividing the space until the specified resolution is reached. The algorithm monitors vehicle movements across this area. When k_{min} different vehicles have visited the area, a time interval $[t_1, t_2]$ is computed as follows: t_2 is set to the current time, and t_1 is set to the time of request minus a random cloaking factor. The area and the time interval are then returned.

6 Implementation

To be effective, the location anonymizer requires location-based services that are used with precise position information by a large user base. We anticipate that such services will soon be available based on telematics, mobile phone, or wireless community network platforms. To our knowledge, no such suitable testbed exists to date. Therefore, we implemented the anonymization algorithms on a Java server platform and evaluated them using automotive traffic simulations based on US geological survey (USGS) cartographic material.

The USGS publishes detailed transportation network information at the city level in the Spatial Data Transfer Standard [33]. We extracted vector coordinates of primary, secondary, and minor roads from the transportation layer of the $1 : 24,000$ scale Digital Line Graph [34] data files. The data has a resolution of 0.61m. Specifically, we selected 2000x2000m areas from the city of

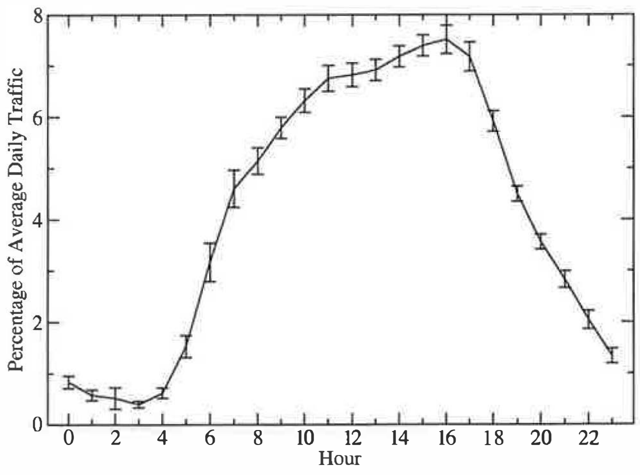


Figure 2: Hourly traffic volume relative to daily traffic volume during a typical August 2002 day

Denver, Colorado, where we had access to traffic count statistics. Figure 1 shows maps of selected areas. The thickest lines indicate expressways, the medium lines arterials, and the thin lines collector streets. Two maps (area 1, area 2) included predominantly expressways, the other maps (area 3, area 4) mostly collector streets. Coordinates are given in meters in zone 13 of the Universal Transverse Mercator (UTM) projection using the North American 83 geodetic datum.

A traffic study [35] reports the 24 hour traffic volume at specific points along roads. We averaged the counts for different urban road types and mapped them onto the USGS road classes as shown in Table 3. Traffic volume was computed as the average 24 hour bi-directional traffic count.

USGS Class	Road Type	Traffic Volume
1	Expressway	70000
2	Arterial	22000
3	Collector	6000

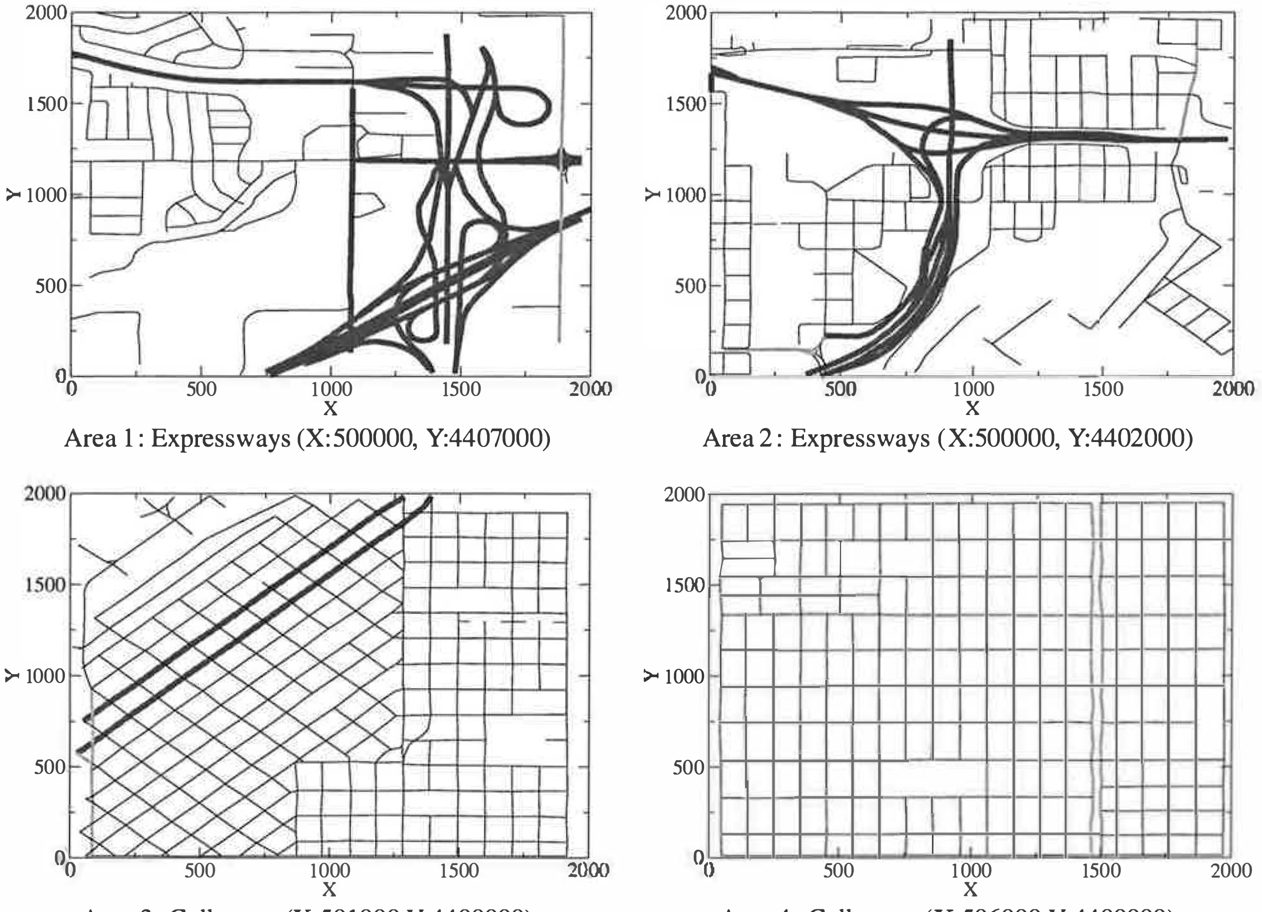
Table 3: Mapping of Traffic Study volumes to road classes from USGS data

The algorithms are evaluated at different times of day, because traffic volume changes heavily between peak and night hours. An adjustment factor for each hour was derived as follows. The Colorado Department of Transportation maintains continuous traffic counters along several highways. For each hour of a day, we calculated the percentage of total daily traffic present during this hour from the mean August 2002 traffic counts for 25 highways [36]. Figure 2 shows the results marked with 95% confidence intervals.

To create a traffic snapshot for a given hour, we place vehicles on the road segments according to a uniform stochastic process. The number of vehicles on a road

```
1. Initialize the quadrants $q$ and $q_{prev}$ as the total area covered by the anonymizer
2. Initialize a traffic vector with the current positions of all known vehicles
3. Initialize $p$ as the position of requestor vehicle
4. If the number of vehicles in traffic vector $< k_{min}$,
   then return the previous quadrant $q_{prev}$
5. Divide $q$ into quadrants of equal size
6. Set $q_{prev}$ to $q$
7. Set $q$ to the quadrant that includes $p$
8. Remove all vehicles outside $q$ from the traffic vector
9. Repeat from Step 2
```

Table 2: Adaptive-interval cloaking algorithm. The algorithm computes an area (quadrant) that includes the actual requester and enough potential requesters to satisfy the anonymity constraint k_{min}.



Area 1: Expressways (X:500000, Y:4407000)

Area 2: Expressways (X:500000, Y:4402000)

Area 3: Collectors (X:501000,Y:4400000)

Area 4: Collectors (X:506000,Y:4400000)

Figure 1: Selected 2000x2000m evaluation areas. The thickest lines indicate expressways, the medium lines arterials, and the thin lines collector streets. Two maps (area 1, area 2) included predominantly expressways, the other maps (area 3, area 4) mostly collector streets. Coordinates are in meters based on UTM Zone 13.

segment n is determined by

$$n = \frac{l \times c \times h}{v}$$

with traffic count c, hour adjustment h, vehicle velocity v, length of a road segment l.

For all experiments we assume an average velocity v of 10m/s and report mean results for a 24-hour period, that is, one snapshot for each hour of day. Unless otherwise stated, the anonymity constraint k_{min} was set to 5, which we intuitively judge as a fair level of anonymity.

7 Accuracy Results

Figure 3 presents an overview of our results. It illustrates the dependency of the resulting spatial resolution on road characteristics and traffic densities. For each of the selected maps and corresponding traffic densities, the median spatial resolution of 10000 simulated LBS requests is shown. In addition, the mean anonymity k, which represents the average number of subjects in the chosen area, is plotted against the second scale (right) on the y-axis.

The median resolution decreases as collector street mileage increases over highway mileage. For the highway areas with their high density of vehicles, the median accuracy is 30 and 65 meters. For the collector areas the resolution decreases to 125 and 250 meters. Interestingly, across all areas the spatial intervals selected by the adaptive algorithms not only have the same anonymity bound (5 subjects), but also a similar mean anonymity at approximately 10 subjects.

Figure 4 and Figure 5 provide more detail on the spatial resolution results by showing the relative distribution of resolutions in the form of histograms for a highway (1) and a collector area (4), respectively. While in the highway area less than 10% of requests reach a resolution lower than 125 meters, it is about 60% for the collector street area. Figure 5 also illustrates the relationship between anonymity and resolution in a single area. For lower resolutions (bigger areas) the mean anonymity does not stay near the minimum, but increases to more than triple the k_{min} constraint of five. When the algorithm is forced to choose a lower resolution, it has to quadruple the area and thereby includes more vehicles than necessary.

Figure 6 illustrates the tradeoff between the degree of anonymity and resolution, showing median resolution and mean anonymity k for different anonymity constraints k_{min}. The results stem from area number 3 with predominantly collector streets. Resolution is negatively correlated to the anonymity constraint. Also note that mean anonymity is approximately double the anonymity constraint. Again, this suggests that an improved discretization algorithm could yield better resolution with lower mean anonymity (i.e., closer to the minimum constraint).

Spatial resolution can also be improved through reductions in temporal accuracy. Figure 7 shows the mean reduction in temporal resolution (and delay of messages) required to reach a specified spatial resolution. Results are reported for a highway area (2) and a collector street area (3). The anonymity constraint is also varied between five and ten. As expected, the temporal accuracy decreases for higher anonymity constraints, more collector streets, and higher spatial resolution. For highways the temporal resolution stays below 30s for resolutions up to 15m. On collector streets the resolution decreases to about 70s at this level of spatial accuracy.

8 Discussion

The analysis concentrates on interpreting the accuracy results and identifying anonymity and security limitations. We define security problems as adversarial attempts to obtain more accurate or extra data from the system that violates the anonymity constraint. Anonymity problems involve identification based on the data allowed by the anonymity constraint.

8.1 Accuracy

The results are encouraging when compared against the E-911 requirements introduced in Section 3 as a yardstick for useful location information. However, they vary widely across different types of areas. The highway areas yield better than required accuracy (less than 10% over

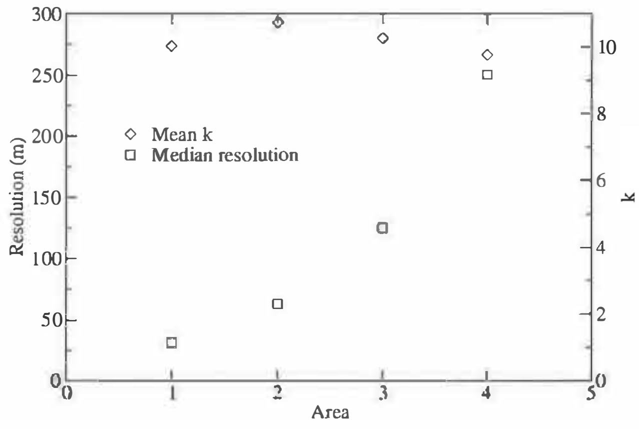


Figure 3: Dependency of spatial resolution and mean anonymity on area characteristics. For each evaluation area, the figure shows the median resolution from a large number of requests (left y-axis scale) and the mean *actual* anonymity—the number of subjects indistinguishable from the requestor (right y-axis scale).

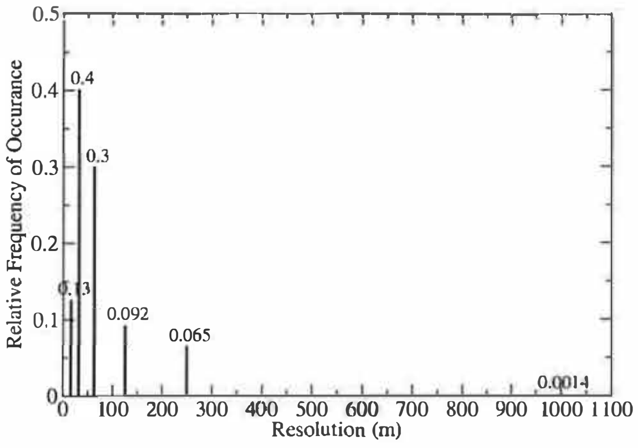


Figure 4: Relative frequency of spatial resolution for highway area (1). This figure illustrates the distribution of resolutions over a large number of simulated requests.

125m), collector street area (3) with a median of 125m is close to the requirements, and collector street area (4) clearly does not meet the requirements.

The results also show that the spatial accuracy can be adjusted through reductions in temporal resolution. The inherent delay of this approach makes it unsuitable for services that require a quick response. However, a large class of monitoring services along the scenarios of driving conditions monitoring and road hazard detection are well served by this approach. Brief delays, comparable to the delays experienced in web browsing over slower Internet connections, might also be acceptable for more interactive LBSs such as a road map service. These delays would also ensure at least 125m resolutions for the collector street area (4). Furthermore, delaying requests becomes unnecessary if the system can precompute temporal and spatial resolutions before the requests are issued. We believe that an investigation of this approach would be a worthwhile continuation of this work.

8.2 Security Analysis

From a security perspective, the wireless carriers or eavesdroppers can attempt to circumvent the location anonymizer and accurately locate a subject using the wireless channel. Authentication and encryption between the location client and the anonymity server effectively prevents them from listening to the exact coordinates or impersonating anonymity servers. The timestamp in the location beacons ensures that the bitstring of subsequent encrypted packets from the same location differs and also protects from replay attacks.

More difficult to prevent are attempts to estimate the location of a transmitter based on physical layer properties of the network. Several cooperating receivers can triangulate the position of a transmitter through methods such as time of arrival (TOA)[29]. Judging from the technical difficulties encountered in implementing the E-911

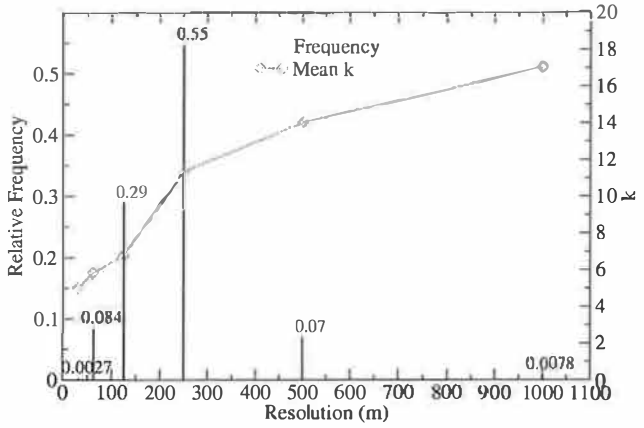


Figure 5: Relative frequency of spatial resolution for collector road area (4). In addition to the distribution of resolutions (left y-axis scale), the figure shows the mean actual anonymity at each resolution (right y-axis scale).

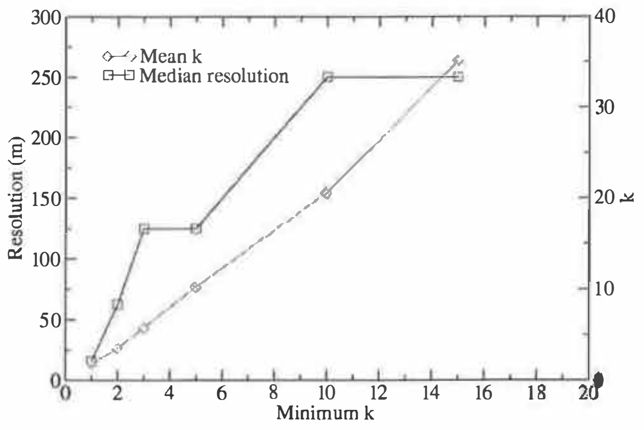


Figure 6: Dependency of spatial resolution and mean anonymity on anonymity constraint. This figure illustrates how spatial resolution (left scale) and mean actual anonymity (right scale) vary with different anonymity constraints (x-axis).

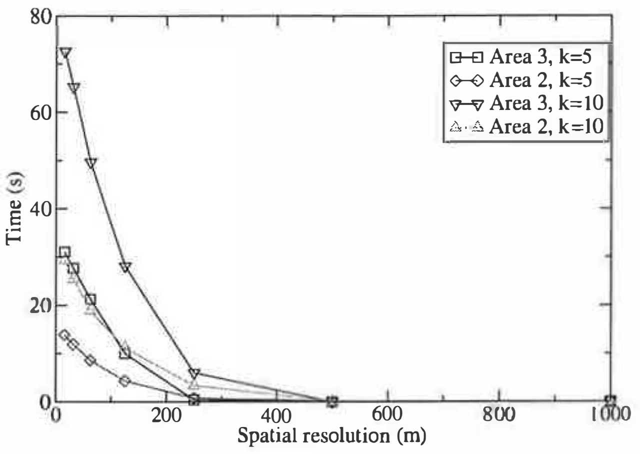


Figure 7: Tradeoff between temporal and spatial resolution. The figure shows the mean reduction in temporal resolution necessary to reach a specified spatial resolution. The tradeoff is shown for a highway (2) and a collector (3) area at different anonymity constraints.

requirement for mobile phones, location information obtained through these mechanisms would likely be about 1-2 orders of magnitude less accurate than information reported by in-vehicle GPS receivers. Thus, anonymity constraints would rarely be violated.

Another potential attack seeks to trick the location server into releasing too accurate data. An adversary could spoof a number of additional *virtual* vehicles or have real vehicles report incorrect location information. If the location server includes these nonexistent vehicles in its computation, the released location information would likely not meet the anonymity constraints. However the location server only accepts one location beacon from each authenticated vehicle. This means, the adversary would need to acquire a potentially large number of authentication keys. Therefore, authentication keys require adequate protection, such as storage in secure hardware.

8.3 Anonymity

k-Anonymity reduces the privacy threats outlined in section 4. If a location tuple is k-anonymous, the adversary cannot uniquely identify the originator of a message through space identification or observation identification, since the tuple matches k 1 other subjects as well. Given no other information the reidentification risk is therefore $\frac{1}{k}$. Similarly, location tracking faces obstacles when attempting to link subsequent location updates to a subject. Since k 1 other subjects are in the area, it is not clear whether the location update actually originated from the same subject. In other words, if multiple subjects are using a LBS through the anonymity service, it is difficult for the LBS to generate movement paths of subjects even if they provide location updates with high frequency.[2]

At this point, it is difficult to gauge which size of k is minimally necessary or sufficient. Fundamentally, it depends on the resources of the potential adversary. A minimum of 2 is obviously required in this particular algorithm to yield any protection. In practice, the parameter will likely be determined through user preferences.

While the basic algorithm ensures k-anonymity for individual location requests, problems can arise when requests for multiple vehicles are issued. Consider the following location tuples obtained from 4 different vehicles:

$$1 : ([0,1], [0,1], [t_1, t_2])$$

$$2 : ([1,2], [0,1], [t_1, t_2])$$

$$3 : ([0,1], [1,2], [t_1, t_2])$$

$$4 : ([0,2], [0,2], [t_1, t_2])$$

These tuples are overlapping in time and space. The first three tuples specify adjacent quadrants, while the fourth one specifies a larger quadrant that covers the three others; this scenario is also illustrated in Figure 8. For simplicity, assume that all tuples were processed with the same k_{min} parameter, say 3, and the time interval is too small for vehicles to significantly move. Then an adversary can conclude that request number 4 must have originated from quadrant $([1,2], [1,2])$, because otherwise the algorithm would have chosen a smaller quadrant. This inference violates the anonymity constraint; it illustrates that an adversary gains information from tuples overlapping in time and space.

Furthermore, sophisticated adversaries may mount an identification attack if they can link multiple requests to the same subject and can repeatedly obtain the subject's

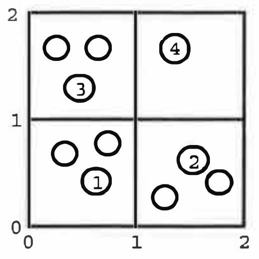


Figure 8: Compromised anonymity through overlapping requests. The circles and squares represent subjects and the quadrants computed by the cloaking algorithm, respectively. If each numbered subject requests a service simultaneously, subject 4 could be identified.

location information from other sources. Consider an unpopular LBS that is rarely accessed. If this service is requested repeatedly from approximately the same spatial region, an adversary could conclude that the requests stem with high probability from the same subject (i.e., link the requests). If, in addition, the adversary knows that a certain subject was present in each of the spatial areas specified in the LBS requests, the adversary can determine with high probability that this subject originated the requests. This inference is based on the assumption that it is unlikely that other subjects from the anonymity set traveled along the same path, given the path is long enough (enough request were observed) relative to the size of the anonymity set. However, such an attack requires a large effort in determining a subject's actual position for a sufficient number of requests.

Although the anonymity constraint is not met in such cases, further research is needed to determine how serious these issues are. In practice, not every overlapping request allows such straightforward inferences and the probability of overlaps depends on the frequency of requests issued by subjects. To ensure meeting the anonymity constraint, disclosure control extensions to the cloaking algorithm could keep track of and prevent overlapping requests. Similarly, the algorithm could take into account the popularity of the accessed LBS to prevent linking of unusual requests.

Finally, it is important to realize that k-anonymity is only provided with respect to location information. Other service-specific information contained inside a message to a LBS could still identify the subject. This is analogous to anonymous communication services, which reduce reidentification risks of network addresses, but do not address other message content. Location information, however, will likely pose more serious risks than other typical message content.

9 Conclusions and Future Work

This paper analyzed the technical feasibility of anonymous usage of location-based services. It showed that location data introduces new and potentially more se-

[2]Recall that we assume subjects do not transmit an identifier or pseudonym such as user ID to the LBS that would allow for trivial linking of subsequent location updates

vere privacy risks than network addresses pose in conventional services. Both the reidentification and the location tracking risk can be reduced through k-anonymous data. A system model and a quadtree-based algorithm were introduced to guarantee k-anonymous location information through reductions in location resolution. The main question we addressed was whether the resulting data accuracy is adequate for location-based services. Since the accuracy is dependent on traffic conditions, the algorithm was empirically evaluated using a traffic distribution model derived from traffic counts and cartographic material. Specifically, we draw the following conclusions:

- The quadtree-based algorithm reached accuracy levels comparable to the phase II E-911 requirements, and thus should be suitable for many location-based services.
- In areas with major highways the median accuracy is approximately 30m and increases to 250m for city areas with large block sizes. These results were obtained with an anonymity constraint of 5, yielding a mean anonymity level of approximately 10 people who may have issued a particular request.
- Spatial resolution can be significantly improved through a several seconds reduction in temporal resolution. Because of the imposed delay, this method is most applicable to noninteractive services.

9.1 Future Work

There are three directions for future work. The first avenue attempts to improve upon the resolution of the anonymizer. We plan to study clustering algorithms that can more intelligently pick minimally sized areas with sufficient traffic. The mean traffic volume in the areas identified by the current algorithms is approximately double the anonymity constraint, which leaves ample room for improvements. Furthermore, the algorithms should be able to operate with incomplete location information, where the position of subjects is periodically sampled rather than continuously updated.

The more difficult issue is decoupling the anonymizer from the current client-server architecture. For individual users to remain anonymous, the location server must have sufficient users within a geographic locale; unless the different users subscribe to the same location service, the reduced sample population available to any given location server may not suffice to anonymize queries for a given area. The algorithms we have used are efficient, and could execute on a wireless device. However, they require location information from different devices in the local area in order to judge the density of devices. Thus, at first sight, a "peer-to-peer" location anonymizing system requires access to the same information that it is attempting to cloak.

Lastly, we plan to deploy this anonymity system in a wireless LAN community network. Such community networks use high-speed wireless networking to provide Internet access; one example are the wireless access points common at coffee shops. These wireless networks have a limited range of 300–1500 feet, meaning that coarse location information can be determined simply by associating with a specific access point. In these networks, location based cloaking must occur at the application, network and physical layers.

Acknowledgments

Paul Chou and our colleagues at the IBM T.J. Watson Research Center encouraged us to research location privacy challenges. The anonymous referees and our paper shepherd, Maria Ebling, provided useful comments on a draft of this paper. This work was funded by award #9988548 from the Division of Computer-Communications Research of the National Science Foundation. All views and opinions stated reflect those of the authors, not those of the National Science Foundation.

References

[1] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The cricket location-support system. In *Proceedings of the sixth annual international conference on Mobile computing and networking*, pages 32–43. ACM Press, 2000.

[2] I. Getting. The global positioning system. *IEEE Spectrum*, 30(12):36-47, December 1993.

[3] Mike Spreitzer and Marvin Theimer. Providing location information in a ubiquitous computing environment (panel session). In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 270–283. ACM Press, 1993.

[4] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The anatomy of a context-aware application. In *Mobile Computing and Networking*, pages 59–68, 1999.

[5] Rui Jose and Nigel Davies. Scalable and flexible location-based services for ubiquitous information access. In *Proceedings of First International Symposium on Handheld and Ubiquitous Computing, HUC'99*, pages 52–66. Springer Verlag, 1999.

[6] C. Bisdikian, J. Christensen, J. Davis II, M. Ebling, G. Hunt, W. Jerome, H. Lei, and S. Maes. Enabling location-based applications. In *1st Workshop on Mobile commerce*, 2001.

[7] P. A. Karger and Y. Frankel. Security and privacy threats to ITS. In *Proceedings of the Second World Congress on Intelligent Transport Systems*, volume 5, Yokohama, Japan, Nov 1995.

[8] Roy Want, Andy Hopper, Veronica Falco, and Jonathan Gibbons. The active badge location system. *ACM Transactions on Information Systems (TOIS)*, 10(1):91–102, 1992.

[9] Philip E. Agre. Transport informatics and the new landscape of privacy issues. *Computer Professionals for Social Responsibility (CPSR) Newsletter*, 13(3), 1995.

[10] Marc Langheinrich. Privacy by design – principles of privacy-aware ubiquitous systems. In G.D. Abowd, B. Brumitt, and S. Shafer, editors, *Ubicomp 2001 Proceedings*, volume 2201 of *Lecture Notes in Computer Science*, pages 273–291. Springer, 2001.

[11] Andreas Pfitzmann and Marit Koehntopp. Anonymity, unobservability, and pseudonymity —a proposal for terminology. In Hannes Federrath, editor, *Designing Privacy Enhancing Technologies — Proceedings of the International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *LNCS*. Springer, 2000.

[12] Michael K. Reiter and Aviel D. Rubin. Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[13] D. Goldschlag, M. Reed, and P. Syverson. Onion routing for anonymous and private internet connections. *Communications of the ACM (USA)*, 42(2):39–41, 1999.

[14] NextBus Information Systems. Nextbus website. 1321 67th Street, Emeryville, CA 94608, USA, `http://www.nextbus.com`, Oct 2002.

[15] J. Cuellar, J. Morris, and D. Mulligan. Internet engineering task force geopriv requirements. http://www.ietf.org/html.charters/geopriv-charter.html, Oct 2002.

[16] Sastry Duri, Marco Gruteser, Xuan Liu, Paul Moskowitz, Ronald Perez, Moninder Singh, and Jung-Mu Tang. Framework for security and privacy in automotive telematics. In *Proceedings of the second international workshop on Mobile commerce*, pages 25–32. ACM Press, 2002.

[17] Jalal Al-Muhtadi, Roy Campbell, Apu Kapadia, M. Dennis Mickunas, and Seung Yi. Routing through the mist: Privacy preserving communication in ubiquitous computing environments. In *Proceedigns of IEEE International Conference of Distributed Computing Systems (ICDCS)*, pages 65–74, Vienna, Austria, Jul 2002.

[18] Asim Smailagic and David Kogan. Location sensing and privacy in a context-aware computing environment. *IEEE Wireless Communications*, 9(5):10–17, Oct 2002.

[19] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[20] A. Pfitzmann, B. Pfitzmann, and M. Waidner. Isdnmixes: Untraceable communication with very small bandwidth overhead. In Wolfgang Effelsberg, Hans Werner Meuer, and Günter Müller, editors, *Proceedings of Kommunikation in Verteilten Systemen, Grundlagen, Anwendungen, Betrieb, GI/ITG-Fachtagung*, volume 267 of *Informatik-Fachberichte*, Mannheim, Germany, Feb 1991. Springer.

[21] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing. *Communications of the ACM*, 42(2):39–41, 1999.

[22] Anonymizer. Anonymizer website. 5694 Mission Center Road #426, San Diego, CA 92108-4380, `http://www.anonymizer.com`, 2000.

[23] Clay Shields and Brian Neil Levine. A protocol for anonymous communication over the internet. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 33–42. ACM Press, 2000.

[24] Yong Guan, Xinwen Fu, Riccardo Bettati, and Wei Zhao. A quantitative analysis of anonymous communications. *IEEE Transactions on Reliability*, (to appear).

[25] Nabil R. Adam and John C. Worthmann. Security-control methods for statistical databases: a comparative study. *ACM Computing Surveys (CSUR)*, 21(4):515–556, 1989.

[26] Rakesh Agrawal and Ramakrishnan Srikant. Privacy-preserving data mining. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 439–450. ACM Press, May 2000.

[27] Dakshi Agrawal and Charu C. Aggarwal. On the design and quantification of privacy preserving data mining algorithms. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 247–255. ACM Press, May 2001.

[28] P. Samarati and L. Sweeney. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. Technical Report SRI-CSL-98-04, Computer Science Laboratory, SRI International, 1998.

[29] J. Reed, K. Krizman, B. Woerner, and T. Rappaport. An overview of the challenges and progress in meeting the e-911 requirement for location service. *IEEE Personal Communications Magazine*, 5(3):30–37, April 1998.

[30] Tele Atlas North America, Inc. Geocode website. 1605 Adams Drive, Menlo Park, CA 94025, `http://www.geocode.com/`, Oct 2002.

[31] J.A. Simpson and E.S.C. Weiner, editors. *Oxford English Dictionary, Second Edition.* Clarendon Press, 1989.

[32] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[33] U.S. Geological Survey (USGS). Spatial data transfer standard. 12201 Sunrise Valley Drive, Reston, VA 20192, USA, `http://mcmcweb.er.usgs.gov/sdts/`, 1995.

[34] U.S. Geological Survey (USGS). Digital line graph data. 12201 Sunrise Valley Drive, Reston, VA 20192, USA, `http://edc.usgs.gov/geodata/`, Oct 2002.

[35] DCROG. Denver regional council of governments: Denver regional travel behavior inventory, 2001. 2480 W. 26th Avenue, Suite 200B, Denver, CO 80211-5580.

[36] Colorado Department of Transportation. Traffic statistics & data. Public Relations Office, 4201 E Arkansas Ave, Denver, CO 80222, `http://www.dot.state.co.us/`, Oct 2002.

Reservations for Conflict Avoidance in a Mobile Database System *

Nuno Preguiça, J. Legatheaux Martins, Miguel Cunha, Henrique Domingos
Departamento de Informática
FCT, Universidade Nova de Lisboa, Portugal

Abstract

Mobile computing characteristics demand data management systems to support independent operation. However, the execution of updates in a mobile client usually need to be considered tentative because uncoordinated updates that conflict need to be reconciled. In this paper we present a mechanism to independently guarantee that updates can be executed in the server without conflicts. To this end, clients obtain leased reservations upon the database state. Updates are specified as common small PL/SQL programs, dubbed mobile transactions, that execute both in the mobile client and in the server. Using the available reservations, the client transparently verifies that a transaction can be executed in the same way both in the mobile client and in the server, thus leading to the same final result. Mobile transactions may specify conflict detection and resolution rules to be used when transactions cannot be locally guaranteed.

1 Introduction

Despite advances in hardware and communication technology, the connectivity in mobile devices is intermittent because of physical, economical and energy factors. These characteristics call for support of independent operation, i.e., users must be allowed to read and modify a shared database with no remote synchronization. Even when connectivity is available, independent operation may be used to overcome latency and bandwidth problems, interference among multiple long interactive transactions and to reduce load during service peaks.

Optimistic replication is used to support independent operation [19, 14, 20, 11]. In such approaches, clients maintain local data copies and modify them independently. The uncoordinated updates are integrated in the common database state solving any conflict caused by concurrent modifications. As this reconciliation process may lead to a different final result (or even to the abortion of the transaction), the client can not independently determine the final result of a transaction.

This paper presents the Mobisnap mobile database system, focusing on support for independent operation. The Mobisnap middleware extends a traditional client/server SQL database system. The single server maintains the *official* database state. Mobile clients cache database snapshots. Applications running on mobile devices update the shared database submitting small PL/SQL [13] programs, dubbed *mobile transactions.* During independent operation, mobile transactions are tentatively executed in the mobile clients. Later, the clients propagate these mobile transactions to the server, where the transaction programs are re-executed against the *official* database state.

Unlike transaction re-execution and validation based on *read and write sets*, the execution of the mobile transaction program in the server allows the definition of conflict detection and resolution rules that exploit the semantics of the operation (a similar approach is used in Bayou [20]).

Mobisnap combines this conflict resolution strategy with a conflict avoidance mechanism based on *reservations.* A reservation provides some kind of promise upon the database state, depending on the type of reservation. Mobile clients may obtain leased [5] reservations before disconnecting. When a mobile transaction is executed in the client, the system transparently verifies if the available reservations are sufficient to guarantee its final result independently. If the client holds enough reservations, the local result can be considered definite because reservations guarantee that no conflict will arise when the transaction is re-executed in the server.

Our reservation mechanism presents the following contributions. First, it includes several types of reservations (unlike other proposals that define a single type, such as escrow techniques [9]). As it is shown in section 4.5, it is usually necessary to combine the use of different types of reservations to guarantee the result of any transaction. Second, the system transparently verifies if the available reservations can guarantee the result of a mobile transaction written in unmodified PL/SQL (unlike previous systems [12], no special function is used to access the reserved data items). Third, it implements the reservation model integrated with mobile transactions in a SQL-based system.

*This work is partially supported by FCT/MCT. Nuno Preguiça is partially supported by a FSE scholarship.

The Mobisnap implementation described in this paper represents an interoperable and evolutionary middleware approach towards mobility, instead of a "revolutionary" one. Application that run in the Mobisnap system can use the new mechanisms – mobile transactions and reservations. However, legacy clients are still allowed to access the database server directly, without modifications.

This paper is organized as follows. Section 2 discusses the design principles in the context of typical applications. Section 3 presents the overall Mobisnap system. Section 4 details reservations. Section 5 describes the current status of our work. Section 6 presents an evaluation of reservations. Section 7 discusses related work and section 8 concludes the paper with some final remarks.

2 Design principles

The main concepts of the Mobisnap system will be highlighted with the help of three typical scenarios — we will come back to these examples throughout the paper. In the first scenario, the user is a mobile salesman selling commodities (e.g., CD's, shoes, etc.). The number of commodities can be large and the available stock for each one is limited — all items of a given commodity are assumed identical. The salesman receives orders from her customers that must be satisfied with the current stock.

The second scenario is a variant of the first one, but the commodity's items are not identical (e.g., tickets for the theater, train tickets, etc.).

In the third scenario, the database contains a datebook with appointments. Several persons can change the same datebook (e.g. a person and his secretary). The operations typically insert or remove an appointment.

In all the above scenarios, the mobile database system must support independent operation to allow users to continue their work while disconnected. During independent operation the system can provide local access to shared data using partial database snapshots. For example, a salesman only needs to cache information about the products he sells and the customers he intends to visit.

2.1 Mobile transactions, a tool for handling concurrent updates

A transaction executed independently in a mobile device observes the cached database state. Later, when the transaction is propagated to the server, other transactions might have changed the database. In this case, the transaction execution in the client, as defined by its *read and write sets*, might no longer be valid. To solve these situations it is necessary to rely on semantic information [20, 11] to define appropriate conflict detection and resolution rules that must be enforced when transactions are integrated in the *official* database state.

In Mobisnap, operations that modify the database are expressed as *mobile transactions* (or simply transactions where no confusion may arise). A mobile transaction is a small program written in a subset of the PL/SQL [13] language. The following changes to PL/SQL have been introduced: (1) *commit* and *rollback* statements end the execution of a mobile transaction, i.e., they act as a return; (2) when the result of a *select into* statement includes more than one (sub-)row, one row is returned (instead of raising an exception); (3) the *newid* function was added to return the same new unique identifier when the transaction is executed in the client and in the server.

A mobile transaction is executed tentatively in the client, and later propagated to the server. Its final result is only obtained when the transaction is executed in the server. A mobile transaction is always executed running its program with a specially designed PL/SQL interpreter. Each mobile transaction accesses the database in the context of a distinct database transaction.

Programmers may reason about a mobile transaction as a mobile program that will be executed in the server. This approach allows programmers to express suitable conflict detection and resolution rules for each situation. The following examples, from the previous application scenarios, show two possible approaches. The mobile transaction of figure 1 inserts a new order received by a mobile salesman. In this case, the conditions for accepting the order are precisely expressed — stock availability and price acceptability are checked (line 7) before inserting the order (lines 8-11). The transaction is not aborted if the values are different in the server, but only if the specified conditions are violated. The mobile transaction of figure 2 inserts a new appointment in a shared calendar giving two alternative periods of time (lines 3-4 and 8-9 check the calendar availability for each alternative). In this case, conflicts can be solved if, at least, one of the expressed alternatives is possible.

2.2 Reservations, a tool for guaranteeing results independently

As it has been explained before, the independent execution of a transaction must be considered tentative. However, independently guaranteeing the result of a transaction might be important. For example, a mobile salesman could immediately guarantee that the orders received can be satisfied. To this end, locks are the traditional solution, but they must be adapted to be used in mobile environments. As a mobile client must keep the locks while it is disconnected, locks should affect as few data items as possible. Sometimes locks are still unnecessarily restrictive. For example, escrow techniques [9] can guarantee concurrent updates to variables that represent partitionable resources – the available resources are divided among the mobile clients.

Any of these solutions alone is insufficient to guarantee

```
 1  -------- ORDER PRODUCT: name = "BLUE THING"; quantity = 10; highest price = 50.00 ------
 2  DECLARE
 3    prd_price FLOAT;
 4    prd_stock INTEGER;
 5  BEGIN
 6    SELECT price, stock INTO prd_price, prd_stock FROM products WHERE name = 'BLUE THING';
 7    IF prd_price <= 50.00 AND prd_stock >= 10 THEN       -- checks price acceptability and stock availability
 8      UPDATE products SET stock = prd_stock - 10 WHERE name = 'BLUE THING';
 9      INSERT INTO orders VALUES (newid,'Clt foo','BLUE THING',10,prd_price);   -- newid returns the same unique identifier
10      NOTIFY( 'SMTP', 'sal-07@thingco.pt', 'Order accepted ...');              -- in the client and in the server
11      COMMIT prd_price;                                  -- commits and returns the price used in transaction
12    ENDIF;
13    ROLLBACK;                                            -- rollbacks and returns when customer's preferences cannot be satisfied
14  ON ROLLBACK NOTIFY( 'SMS', '351927435456', 'Impossible order...');    -- sends notification to user on rollback
15  END;
```

Figure 1: Mobile transaction specifying a new order submitted by a mobile salesman.

```
 1  ----------SCHEDULE MEETING: '17-FEB-2002' at 10:00 OR '18-FEB-2002' at 9:00 --------------------------------
 2  BEGIN
 3    SELECT count(*) INTO cnt FROM datebook WHERE day='17-FEB-2002' AND hour=10;        -- first alternative
 4    IF (cnt = 0) THEN                                  -- checks calendar availability for 17-FEB-2002 at 10:00
 5      -- code omitted: update datebook, send notification if appropriate, ...
 6      COMMIT ('17-FEB-2002',10);                       -- commits and returns info on the committed alternative
 7    ENDIF;
 8    SELECT count(*) INTO cnt FROM datebook WHERE day='18-FEB-2002' AND hour=9;         -- second alternative
 9    IF (cnt = 0) THEN                                  -- checks calendar availability for 18-FEB-2002 at 9:00
10      -- code omitted: update datebook, send notification if appropriate, ...
11      COMMIT ('18-FEB-2002',9);                        -- commits and returns info on the committed alternative
12    ENDIF;
13    ROLLBACK;                                          -- rollbacks and returns when no alternative is possible
14  ON ROLLBACK NOTIFY( 'SMS', '351927435456', 'Impossible to schedule... ');
15  END;
```

Figure 2: Mobile transaction adding a new appointment to a shared calendar (declaration of variables is omitted).

the outcome of the mobile transactions presented in figures 1 and 2 (without being too restrictive). Consider the example of figure 1. To guarantee that the order can be accepted, it is necessary to guarantee that the conditions expressed in line 7 are true when the transaction is executed in the server. Escrow techniques can guarantee stock availability ($prd\_stock \geq 10$), but they do not help to guarantee price acceptability ($prd\_price \leq 50.00$). In this case, a lock can be used but it is too restrictive (a reservation that allows to use a reserved value for a data item, despite of its current value, can be used in our system, as described in section 4). Application knowledge, external to the system, can also be used to guarantee the validity of some conditions by voluntarily limiting the degree of concurrency (e.g. prices are only updated during the night). This approach is not safe and it can be easily broken in a large scale scenario. Therefore, the system needs to combine different techniques in a single framework to safely guarantee the outcome of a transaction independently (see other examples in section 4.5).

In Mobisnap, the outcome of mobile transactions can be guaranteed independently using an SQL-based *reservation* mechanism (detailed in section 4). A reservation is similar to a semantic lock and it provides some promise upon the database state. Mobisnap defines several types of reservations, each one providing a different type of promise. The different types of reservations were introduced to support the typical scenarios described earlier. Reservations are valid during a limited period of time (thus preventing a client that becomes permanently disconnected from holding a reservation forever).

Mobile clients obtain reservations before disconnecting. When a transaction is executed in the client, the system transparently checks that the client holds enough reservations to prevent other clients from making updates that might later be found to conflict with the local transaction. If enough reservations exist, the client can independently guarantee the final result of the transaction because reservation guarantee that the transaction program is executed in the same way both in the client and in the server (given that it is propagated to the server before the used reservations expire).

Our reservation model transparently uses all available reservations to guarantee mobile transactions (i.e., the transaction do not need to specify which reservation should be used). Thus, no special function is needed to manipulate the reserved data items and programmers can specify operations as normal PL/SQL programs.

Some ideas used in our reservation model had already been proposed [5, 9, 7]. However, the integration of multiple types of reservations (including the introduction of new ones) and their adaptation to mobile environments, the transparent detection of their usage and its implementation in an SQL-based system are new contributions that are important for the usability of these concepts. As far as we know, its integration with mobile transactions makes our system unique.

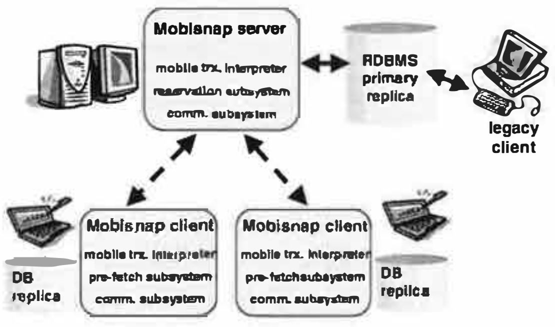


Figure 3: The Mobisnap system architecture.

3 Mobisnap system

The Mobisnap system manages information structured according to the relational data model. Its architecture is based on an *extended client/server* middleware architecture, as depicted in figure 3. Mobisnap server and clients rely on unmodified relational database systems to store data. The server holds the primary copy of all data items in the central database. The server is expected to be mostly available and well connected. Legacy clients can access the central database directly (i.e., without accessing the Mobisnap server).

Mobisnap clients (or simply clients) can run on mobile or stationary computers and be disconnected for long periods of time. Mobisnap clients do not access the central database directly – they always communicate with the Mobisnap server. Users access the database using applications that run on client machines. When connected, the read and write operations executed by an application may be immediately executed in the server.

We now describe independent operation (and the mechanisms to support it). A mobile client locally replicates a subset of the database. This partial replica contains a subset of the database tables; for each table a subset of the columns; finally, only a subset of the records is cached for each table. Currently, the user/application defines the cache contents using a variant of the SQL *select* statement. Other data accessed by the application is also cached using a least recently used algorithm. The prefetch subsystem in the client periodically updates cache contents. This simple strategy could be improved using more complex caching techniques (e.g. [3, 10]), but this problem is outside of the scope of this paper.

The client maintains two copies of the database state: a tentative and a committed one. The tentative version contains the state after executing all disconnected transactions. The committed version contains the state after executing transactions that are guaranteed by reservations. This version is only maintained if enough resources exist. Both versions are refreshed when a mobile clients connects to the server by obtaining, at least, a new copy of the modified records. Applications may read either version using SQL queries — the application selects which version should be used for each query.

Applications modify the database by submitting mobile transactions. The client executes each mobile transaction locally and returns a result. If the client holds enough reservations to guarantee the outcome of the transaction, the result can be considered definite. In this case, both versions of the cache are updated. Otherwise, the result is tentative – only the tentative version is updated.

After being locally executed, a mobile transaction is logged in the mobile device. When the mobile client reconnects, it propagates logged transactions to the server. On weakly-connected environments, propagation may be incremental.

When the server receives a mobile transaction, it executes the transaction program to obtain its final result. If the transaction has been guaranteed in the client and all used reservations are still valid, the reservation model guarantees that the outcome of the transaction is the same in the client and in the server. In the next section we detail the reservation model and its interaction with transaction processing.

When a client obtains a reservation, the server must guarantee that no transaction from any other client violates the promise provided by the reservation. For example, if a client reserves one seat in a train (to guarantee transactions independently), the server guarantees that, at least, one seat remains available – any transaction that tries to book the last available seat is aborted. If the reservation is not used, some transactions may have been aborted unnecessarily – in the end, one seat is available.

To handle these situations, Mobisnap includes a reevaluation mechanism. Transactions that abort in the server can be re-executed after relevant reservations are cancelled or expire. In this sense, a reservation can be seen as an option to modify the database first. When an application submits a mobile transaction, it may specify that the reevaluation mechanism should be used and set the deadline to obtain the final result, i.e., the last time for transaction re-execution in the server.

When the final result of a mobile transaction is obtained, the user may no longer be connected to the system. In Mobisnap, mobile transactions may use the *notify* function to immediately notify users of its final result (e.g. using SMS or pager messages). The *notify* invocations are handled in a special way: they are only executed during the final execution of the transaction in the server. The messages produced by the *notify* function are propagated asynchronously by the Mobisnap server (that handles temporary errors). A similar approach can be used to defer other actions on the outside world until the definite execution of a transaction, thus avoiding the need to undo the effects of tentative actions.

4 Reservations

The goal of reservations is to independently guarantee the final result of mobile transactions by guaranteeing that no conflict will arise when the transaction program is executed in the server. The Mobisnap reservation model is designed to achieve this goal while applications continue to use traditional SQL statements.

4.1 Types of reservations

A reservation provides some kind of guarantee for the future execution of a transaction in the server. In this subsection we present the types of reservations implemented in Mobisnap (see section 4.5 for examples).

Value-change and slot reservations

A *value-change reservation* provides the exclusive right to modify the state of an existing data item (i.e., a subset of columns in some row). For example, a user may reserve the right to change the description of a specific seat in a train. This is like a traditional fine-grain write lock [7].

A *slot reservation* provides the exclusive right to insert/remove/modify rows with some given values. For example, a user may reserve the right to schedule a meeting in a given room in some defined period. This is like a predicate lock [7], and includes rows that exist and that do not exist.

The granularity and lease time of these reservations must be selected carefully because they prevent other transactions from modifying the selected data. In this sense, these reservations can be seen as a mechanism to move the primary copy of a data item to a mobile device.

Value-use reservation

A *value-use reservation* provides the right to use a given value for some data item (despite its current value). The need for this kind of guarantee is common in real life – e.g. a mobile salesman can guarantee a price for an order despite concurrent changes. This reservation implements this idea, allowing transactions to guarantee the value of a data item without unnecessarily restricting concurrent changes. Transactions that use this reservation *observe* the reserved value of the data item (*select* statements return the reserved value instead of the current value).

Escrow reservation

An *escrow reservation* provides the exclusive right to use a share of a partitionable resource represented by a numerical data item. For example, the stock of some product may be split among several salesmen. This reservation is based on the escrow model [12, 9].

The escrow model is applied to numerical data items that represent partitionable resources, where all items are identical — e.g. the number of CDs in the stock, x (in section 4.5, we show how reservations can address problems with non-identical resources – e.g. seats in a train). The following properties usually hold for this type of data items. First, they are updated by adding/subtracting a constant value — e.g. when k CDs are sold, we do $x \leftarrow x - k$. Second, some constraints must be maintained — e.g. the stock must be larger than *min*, i.e., $x \geq min$.

For such data items (called escrowable data items), it is possible to partition the resources among several independent replicas — e.g. let x be partitioned in n parts $x_i, 1 \leq i \leq n$, such that $x = \sum x_i$. Each replica has an associated local constraint that guarantees the validity of the global constraint, $x_i \geq min_i : min = \sum min_i$. Each replica may independently guarantee the result of transactions that comply with the local constraints, i.e., it is possible to guarantee the result of transactions that subtract up to $x_i - min_i$ (aggregated value over all transactions in replica i) to x_i.

Example: Let the current stock of CDs be ten ($x = 10$) and the minimum stock be two ($x \geq 2$). Using the escrow model, the current stock can be partitioned between two replicas – e.g. the first replica gets six CDs ($x_1 = 6$) and the second replica gets four CDs ($x_2 = 4$). The global constraint must also be partitioned between the replicas – e.g. the stock must be greater or equal to one in both replicas ($x_1 \geq 1 \wedge x_2 \geq 1$). The first (resp. second) replica can guarantee transactions that subtract up to five (resp. three) CDs from the stock ($x_1 - min_1 = 6 - 1 = 5$ and $x_2 - min_2 = 4 - 1 = 3$).■

We now discuss the Mobisnap implementation of the escrow model. To recognize the aspects involved, consider the example of figure 1. In this mobile transaction, escrow techniques can be used to guarantee the stock availability. The following steps are taken to subtract a constant to the stock (these steps represent a pattern to access escrowable data items): (1) the current value is read and the validity of the operation is checked (lines 6–7); (2) the value of the data item is updated to reflect the executed operation (line 8).

The first aspect to address is to guarantee the validity of the operation (step 1). To this end, it is usual to use an *if* statement that verifies if the update violates the local constraints or not. To use the same condition without any special function, the same local constraints must be defined in the server and in the client. The escrow model, as defined before, cannot be implemented immediately as it uses different constraints in different replicas.

In Mobisnap, we have introduced the following change to the model presented before: $x : x \geq min$ is partitioned in n parts $x_i : x_i \geq min \wedge x - min = \sum(x_i - min)$. As before, each replica may independently guarantee the result of transactions that comply with the the local constraints. However, the same constraints should be enforced in all replicas, thus allowing mobile transactions to verify the validity of an update using the well-known global constraints (or, even, using more restrictive constraints).

Example: To obtain the same guarantees as in the previous example, the following values are used: $x_1 = 7, x_1 \geq 2$ and $x_2 = 5, x_2 \geq 2$. The first (resp. second) replica can guarantee transactions that subtract up to five (resp. three) CDs from the stock ($x_1 - min = 7 - 2 = 5$ and $x_2 - min = 5 - 2 = 3$).

Let x_1 be the server replica and x_2 be the mobile client replica. $x_2 = 5, x_2 \geq 2$ means that the mobile client has reserved the right to subtract 3 to x (i.e., to guarantee orders for 3 CDs). We say that the mobile client has obtained an escrow reservation for three instances of x. In the server, the value of x is updated ($x_1 = 10 - 3 = 7$) to reflect the guarantees obtained by the client. Thus, no transaction from any other client can use the reserved resources. Alternatively, the value of x_2 can be seen as the minimum value of x when the transactions from the mobile client are executed in the server (i.e., $x \geq x_2$). Therefore, the value of x_2 can only be used to guarantee conditions that use similar relational operators – e.g. the condition $x \leq 10$ can not be guaranteed by the escrow reservation $x_2 = 5, x_2 \geq 2$. The value of x_1 can be seen as the value of resources not reserved by any mobile client (and thus available, for example, to legacy clients).■

The second aspect to address is the *real* update of the escrowable data item (step 2). Usually, mobile transactions update these data items using the *update* statement. In Mobisnap, the system automatically infers the amount of reserved resources used from the *update* statements. As expected, the used resources are consumed from the escrow reservation, i.e., the following transactions can only be guaranteed by the remaining reserved resources.

As described, our approach is completely transparent for programmers. Programmers write mobile transactions with no reference to reservations. If the client holds some reservations, the system automatically uses them to guarantee the result of transactions. The system also keeps track of used reservation automatically.

Our current prototype has a limitation: it only allows a single constraint over each escrowable data item (either $x \geq min$ or $x \leq max$) — we anticipate this to be the typical scenario.

Shared value-change and shared slot reservations

A *shared value-change reservation* provides the guarantee that it is possible to modify the state of an existing data item (i.e., a subset of columns in some row). For example, this reservation can be used to guarantee increment operations to a shared counter.

A *shared slot reservation* provides the guarantee that it is possible to insert/remove/modify rows with some given values. For example, this reservation can be used to guarantee operations in append-only tables.

	vc	s	vu	e	shvc	shs
value-change (vc)	no	no	yes	no	no	no
slot (s)	no	no	yes	no	no	no
value-use (vu)	yes	yes	yes	yes	yes	yes
escrow (e)	no	no	yes	yes*	no	no
shared value-change (shvc)	no	no	yes	no	yes	yes
shared slot (shs)	no	no	yes	no	yes	yes

* Yes, while aggregate reservations do not violate global constraint

Table 1: Reservation compatibility table.

These reservations do not provide any promise about the future state of the database. Instead, they prevent other clients from setting exclusive reservations that forbid the execution of the operations. As the examples of section 4.5 show, these reservations are important to fully guarantee the result of a transaction.

4.2 Granting and enforcing reservations

We now describe how reservations are granted and what is done to guarantee that no transaction violates those reservations (including transactions executed directly in the database by legacy clients). In the next subsections, we present the mobile transaction processing in the client and in the server when reservations are used.

A mobile client requests a set of reservations from the server before disconnecting. This set must be defined based on the operations the user is expected to execute until the next interaction with the server. The deduction of good values for this problem can be seen as an extension of the cache hoarding problem [10] (where clients must pre-fetch the data that will be accessed by users) and it is out of the scope of this paper. To solve this problem, forecasting techniques [4] can be used.

In the experiments presented in section 6 we use simple strategies that lead to good results in a mobile sales application. The GUI of our sales application allows to modify the reservations to obtain using a simple form. Internally, reservation requests are submitted using variants of the SQL select statement (e.g. *get value-use reservation price from products where id='5'* requests a value-use reservation for the *price* of product with *id='5'*).

When a reservation is requested, the server checks if it is possible to grant the reservation. First, it verifies if there is no other granted reservation that conflicts with the request. Table 1 presents the compatibility of two reservations that overlap, where *yes* means that it is possible to grant a reservation of a certain type even if other reservations of the other type already exist for the same data item. If data items do not overlap, reservations do not conflict. Second, it verifies if the user of the mobile client can obtain the requested reservations. This verification is based on authorization rules, set by the database administrator, that specify which reservations each user can obtain and for how long.

When a reservation is granted, the Mobisnap system has to guarantee that its underlying promise is not broken by other transactions. To enforce the promises, while allowing legacy clients to directly access the central database, the following actions are executed on the database (note that these actions are reversed to allow the execution of mobile transactions that use the given reservation):

- For each *value-change reservation*, a trigger is set to prevent transactions from modifying the reserved data item. A trigger prevents an update by throwing an error. We say that the update was blocked.
- For each *slot reservation*, a trigger is set to prevent transactions from inserting/deleting/modifying rows with the given values.
- For *escrow reservations*, the value of the escrowable data item is updated as explained earlier.
- For *value-use, shared slot and shared value-change reservations*, no action in the database is needed.

For value-change and slot reservations there is an additional option that can be used: the reservation request may specify a temporary update to reflect the impossibility to change the reserved data item while the reservation is valid. For example, assume that a row represents a seat in a train. When a mobile client obtains a value-change reservation over that row, the reservation may temporarily set the column that indicates if the seat is occupied to true in the server. This change indicates that no unguaranteed transaction should use that seat. If they do, the trigger aborts them, as in the normal case.

The Mobisnap system keeps track of all granted reservations to check conflicting reservation requests and validity of reservation usage. When reservations expire or are cancelled explicitly by the mobile client, the actions executed to enforce reservations are undone.

As it has been said, a reservation is leased [5], i.e., it is only valid for a limited period of time. This property guarantees that restrictions imposed to other transactions to enforce the promises associated with a reservation do not last forever, even if the mobile client that holds the reservation is destroyed or becomes permanently disconnected. On the other hand, the guarantee provided in a mobile client respecting some mobile transaction is only valid if the transaction is propagated to the server before the used reservations expire.

4.3 Transaction processing in the client

The mobile client executes a mobile transaction in, at most, two steps. In the first step, the system executes the transaction program verifying if its outcome can be guaranteed by the available reservations (see details next). If it can, the committed and the tentative database versions are updated. If not, the system rollbacks any change to the database and proceeds to the second step.

In the second step, the system executes the transaction program against the tentative database version. The result of this execution is considered tentative. If the program runs until a *commit* statement, the result is *tentative commit*. If the program runs until a *rollback* statement, the result is *tentative abort*.

If during transaction processing (in both steps) some non-cached data item is needed to proceed (e.g., an *if* statement uses the value of a non-cached column), the execution is aborted, and the result is *unknown*.

Verifying if a transaction can be guaranteed

Now, we outline how the client interpreter verifies if a mobile transaction can be guaranteed. The basic idea consists in running the transaction program and verify every statement in the execution flow.

For each variable, the interpreter maintains not only its current value but also the information if the value is *guaranteed* and the reservations that guarantee it (if any). The value of a *variable is guaranteed* if it was set in a guaranteed SQL read statement, or in an assignment statement that does not include any unguaranteed variable.

An SQL statement can be guaranteed if: (1) the variables used in the SQL statement (as inputs), if any, are guaranteed; (2) there is a reservation that is compatible with the condition expressed and includes the read/written columns. This second prerequisite is verified comparing the semantic description of the reservation with the conditions expressed in the SQL statement (a similar idea is used in semantic caching [3]). The following additional restrictions apply. A *value-use* reservation can not guarantee an *SQL write statement* (*update*, *insert* or *delete*). A shared reservation can not guarantee an *SQL read statement* (*select*).

When an *SQL write statement* is executed, both database versions are updated. A guaranteed *SQL read statement* returns the reserved value. An unguaranteed *SQL read statement* returns the value of the tentative database version.

An *if statement is guaranteed* if all variables involved in the *if* condition are guaranteed (the restrictions explained in section 4.1 apply to variables guaranteed by escrow reservations). If the condition cannot be guaranteed, its value is assumed false, by default. Thus, it is possible to guarantee an alternative update (in a sequence of alternative updates guarded by *if* statements) when the preferred one cannot be guaranteed. For example, in the transaction of figure 2, if the client only holds reservations for the day "18-FEB", it is impossible to guarantee the first alternative (day "17-FEB") but it is possible to guarantee the second one (day "18-FEB"). When the transaction is re-executed in the server, the preferred alternative may be possible. By default, the server executes the same execution path, despite the current value of the database. The application may override the default options and request either to abort client execution if a condition cannot be guaranteed or to execute the first possible alternative in the server, instead of the guaranteed one.

If the program runs until a commit statement, its result is *reservation commit*. In this case, the transaction result is said locally guaranteed. However, depending on the statements that have been guaranteed, the following levels of guarantees are provided.

- *Full*, if all statements in the execution path are guaranteed by reservations. Note that even in this case, it is incorrect to apply, in the server, the *write set* obtained during the client execution because escrowable data items have to be updated using add/remove operations.
- *Read*, if all statements but write statements are guaranteed by reservations. When the transaction is executed in the server, non-guaranteed writes may be blocked by slot or value-change reservations.
- *Pre-condition*, if all executed conditions (*if* statements) are guaranteed. In this case, the system only guarantees that the execution of the transaction program in the server will follow the same execution path.
- *Alternative pre-condition*, if, at least, one condition (*if* statements) could not be guaranteed. In this case, the application has the option to force the same execution path or not (as explained earlier).

As exemplified in section 4.5, an application should use additional domain-knowledge to interpret the meaning of these levels of guarantees (e.g. the *read* and *full* levels are identical if it is known that no reservation will be granted that block the transaction's writes) and to present the result of transactions to users.

If the mobile transaction runs until an *rollback* statement, the system rollbacks the execution and proceeds transaction processing in the second step described in the beginning of this subsection.

When the result of a transaction is *reservation commit*, the mobile client automatically associates the information about program execution (reservations used and the execution path) with the transaction. The client propagates this information to the server. The server uses this information when it executes the transaction, as explained in the next subsection.

4.4 Transaction processing in the server

The execution of a mobile transaction in the server consists in the execution of the transaction program against the central database. If the transaction was not guaranteed in the client, user's intents must be enforced by the conflict detection and resolution rules specified in the code of the transaction. Blocked write statements can be handled by trapping the thrown error in the code of the transaction. Otherwise, the transaction is aborted and the application may check the cause. Aborted transactions may be re-executed using the reevaluation mechanism.

If the transaction has been guaranteed in the client and it has associated reservations, the interpreter that runs the mobile transaction must execute the following additional actions. Before starting to run the program, the system undoes the actions that enforce the used reservations (see section 4.2). For each value-use reservation, the current value of the data item is replaced by the reserved value – when the execution of the transaction ends, the current value is restored in the database.

During the execution of the transaction, for any read statement that was guaranteed in the client by a value-change or slot reservation, the interpreter returns the same value read in the client. This property guarantees that the same rows are read when the solution for a read statement is a set of rows (i.e. it establishes an order on the result set).

The execution of a mobile transaction partially consumes escrow reservations (e.g. if the client had the right to subtract 3 to x, and this transaction subtracts 1 to x, the client remains with the right to subtract 2 to x). All other reservations remain valid. After the execution of the transaction, the system redoes the actions that enforce the used reservations that remain valid.

4.5 Examples

The mobile transaction presented in figure 1 represents a typical transaction executed by a mobile salesman: the insertion of a new order. In figure 4 we present the reservations obtained by the mobile salesman. The following information is maintained for each reservation: a unique identifier; the *type* of reservation; the *columns* that are reserved in the rows identified by the *table* and *condition*; the *value* of the reserved data item; and additional reservation-specific information.

When the transaction is executed in the client, it is obvious that both reservations are necessary to guarantee that the condition expressed in the *if* statement (line 7) is true. The escrow reservation guarantees that $prd\_stock \geq 10$ will be true when the transaction is executed in the server ($prd\_stock_{clt} = 15$, thus $prd\_stock \geq 10$ evaluates to $15 \geq 10 \equiv true$). In fact, the reservation is more general and it gives the right to subtract 15 to the value of prd_stock with the global constraint $prd\_stock \geq 0$ (or the promise that when the transaction is executed in the server, $prd\_stock \geq 15$). The value-use reservation guarantees that the price is acceptable. The update of the stock (line 8) is guaranteed by the escrow reservation.

id	type	table	column	condition	value	info
45-1	*escrow*	products	stock	name='BLUE THING'	15	≥ 0
45-2	*value-use*	products	price	name='BLUE THING'	44.99	-

Figure 4: Reservations obtained by a mobile salesman.

This statement is used to infer the escrowable resources consumed by the transaction — 10 in this case (the number of resources still available after the execution of this transaction is 5 — this is the new value of the reserved data item $(products, stock)$). The *insert* statement (line 9) is not guaranteed by any reservation. Thus, the transaction will have the *read* level of reservation commit. If the application knows that no slot reservation will block the *insert* in the "orders" table, it can be sure that the transaction will succeed in the server.

It is also possible to get additional reservations that guarantee the success of the *insert* statement. First, a shared slot reservation could be obtained for the orders table (uniqueness of identifier in the table is assured by the properties of the *newid* function). Second, an additional column could be added to the orders table specifying the salesman responsible for the order. In this case, the salesman could obtain a slot reservation for his orders.

The mobile transaction presented in figure 5 represents a typical scenario for using non-identical resources. In this example each seat is unique. Therefore, after verifying that there is, at least, one seat left (line 5), it is necessary to obtain the identifier of the seat (lines 7-8) and update the seat information with the name of the new passenger (lines 9-10). Assume that two seat are reserved: the correspondent reservations are presented in figure 6. Up to the need of obtaining the identifier of the seat, the transaction processing can be guaranteed as in the previous example. The *select* statement (lines 7-8) that obtains the seat returns one of the reserved seats (e.g. "4A"). The *update* statement (lines 9-10) updates the seat information with the name of the passenger and the price of the ticket. These statements are guaranteed by the value-change reservation obtained over the seat (remember that, in the client, read statements return reserved data items, if any exists). When the transaction is executed in the server, Mobisnap guarantees that the *select* statement returns the same record (i.e. seat "4A"). These reservations guarantee the *full* level of reservation commit (all statements are guaranteed).

If the value-change reservation was not available, it would be impossible to guarantee the read and write statements of lines 7-10. In this case the transaction would have the *pre-condition* level of reservation commit. For this transaction, this result means it is possible to guarantee the availability of one ticket but, in the client, it is impossible to know which seat will be assigned to the passenger (the *select* statement of line 7-8 would return a tentative result based on the tentative cache version).

The mobile transaction presented in figure 2 represents a typical transaction executed in a shared calendar. The client may obtain a slot reservation to schedule appointments during the morning of day '17-FEB-2002', as shown in figure 7. This reservation guarantees the result of the condition of line 4. The variable *cnt* is guaranteed by the slot reservation because the records selected in line 3 are a subset of the records associated with the slot reservation. The code omitted in line 5 would typically insert a record in the datebook for the given time – this operation can also be guaranteed by the slot reservation. Thus, the result of the transaction would be the *full* level of reservation commit.

These examples show that the combination of multiple reservations is fundamental to guarantee the result of a mobile transaction independently. Note that not even the example that uses anonymous resources (figure 1) can be guaranteed by escrow reservations alone.

5 Status and future work

We have implemented a prototype of the Mobisnap system as a middleware layer in Java. In the server, an Oracle 8 database stores the primary copy of the data, but any database that supports triggers could be used. In the client, we use the Java-based Hypersonic SQL database engine to store the local data copies. Therefore, Mobisnap clients should run in any mobile device that supports Java 2. Until now, we have tested the clients in PCs running Windows and Linux and in Compaq iPAQ handheld computers running SavaJe OS. The Mobisnap middleware runs on top of the RDBMSs to implement the system specific functions, including, reservation management and mobile transaction processing (the database engines are used only to process SQL statements).

In our current prototype, the PL/SQL interpreters have several limitations, mainly the client interpreter that verifies if a transaction can be guaranteed. For example, although sub-queries can be processed, if they are used in a transaction, the transaction cannot be guaranteed. A complete implementation of an PL/SQL interpreter should allow to verify if any deterministic transaction can be guaranteed or not. Non-deterministic functions, such as querying the local time, can be handled by saving the value returned in the client and returning the same value in the server (similar to the *newid* function).

Other issues of the prototype are still being worked out, namely security, some reservation options such as reservation delegation between clients and improved caching management. In the future we intend to evaluate the im-

```
------------ BUY 1 TICKET: train = "London-Paris 10:00"; day = '18-FEB-2002'; price <= 100.00 ------------
BEGIN
  SELECT price, available INTO tkt_price, tkt_avail FROM trains
                        WHERE train = 'London-Paris 10:00' AND day = '18-FEB-2002';
  IF tkt_price <= 100.00 AND tkt_avail >= 1 THEN            -- checks seat availability
    UPDATE trains SET available = tkt_avail - 1 WHERE train = 'London-Paris 10:00' AND day = '18-FEB-2002';
    SELECT seat INTO tkt_seat FROM tickets
              WHERE train = 'London-Paris 10:00' AND day = '18-FEB-2002' AND used = FALSE;  -- get one available seat
    UPDATE tickets SET used = TRUE, passenger = 'Mr. John Smith', price = tkt_price
                 WHERE train = 'London-Paris 10:00' AND day = '18-FEB-2002' AND seat = tkt_seat;
    COMMIT (tkt_seat,tkt_price);                              -- commits and returns thicket information
  ENDIF;
  ROLLBACK;                                                   -- rollbacks when there is no seat available
END;
```

Figure 5: Mobile transaction reserving a new ticket in a train reservation system (declaration of variables is omitted).

id	type	table	column	condition	value	info
37-8	*escrow*	trains	available	train='London-Paris 10:00' AND day='18-FEB-2002'	2	≥ 0
37-9	*value-use*	trains	price	train='London-Paris 10:00' AND day='18-FEB-2002'	95.00	-
37-10	*value-change*	tickets	*	train='London-Paris 10:00' AND day='18-FEB-2002'	$(4A, false, \ldots)$	-
37-11	*value-change*	tickets	*	train='London-Paris 10:00' AND day='18-FEB-2002'	$(4B, false, \ldots)$	-

Figure 6: Reservations obtained in the train reservation system example: two seats are reserved.

Name	***small***	***large***
Duration (t)	12 hours	
Link failure interarrival mean	Exp(120 min)	
Link failure duration	Exp(36 min)	
Message latency	Exp(400 ms)	
Server failure interarrival mean	Exp(6 hours)	
Server failure duration	Exp(1 min)	
Number of clients	8	50
Initial stock ($stock_{init}$)	300	30000
Order quantity of each request (value of request)	round(0.5 + + Exp(1.5))	round(0.5 + + Exp(19.5))
Name	***good***	***bad***
Expected usage rate (e_u)	variable	
Prediction reliability ($sigma_{base}$)	0.04	0.64
Predicted vs. real conformance	$\approx$ 10% in *small* $\approx$ 5% in *large*	$\approx$ 55%

Table 2: Experimental parameters.

pact of reservations (and associated triggers) on the performance of the database server.

6 Evaluation

In this section we evaluate the effectiveness of reservations to support a mobile sales application through simulation. Due to space limitation we can only present a small part of the studied scenarios [17].

The mobile sales application maintains information about a set of products, including for each product, its current stock and price. A mobile salesperson uses a mobile device to submit requests from her customers. In the experiments presented in this paper, a single type of request is used: place a new order. Each new order is submitted as a mobile transaction identical to figure 1. To guarantee the result of these mobile transactions independently, mobile devices obtain reservations.

The experiments simulate the execution of the Mobisnap system, including the single server, a set of mobile clients and the network according to the parameters presented in table 2. Server failure parameters lead to 99.7% availability.

A network module simulates the communications between the server and the clients. We have modelled a simple mobile environment scenario where clients remain disconnected for long periods of time. End-to-end unavailability is 30%. We simulate end-to-end partitions using (client-server) link failures. However, we make no assumption on the cause (e.g. voluntary disconnection, energy restrictions, etc.) of each failure. Message latency has an exponential distribution to (roughly) model variable message delay. Latencies over 1 second are considered as message losses.

Our experiments model two deployment scenarios: a *small* and a *large* scale deployment of the mobile sales application (see table 2). The large scale scenario is used mainly to evaluate the influence of scale. We now explain the parameters for the *small* scale scenario.

In the *small* scenario, there are only eight salespersons (or mobile clients). For simplicity (and without loss of generality), we consider that there is just one product available, the initial stock is 300 and its price is 1. The quantity involved in each order is variable and based on an exponential distribution, as described in table 2 – this approach leads to generally small orders and rare large orders (overall average quantity is approximately 2). As the price of one instance of the product is 1, the value of each request is equal to the quantity in each order.

The requests are generated in mobile devices as follows. First, for each experiment, the expected usage rate, e_u, controls the expected value of all received requests, exp_{total} : $exp_{total} = e_u \times stock_{init}$. In our experiments, e_u varies from 55% to 175% modelling from weak to very strong demands.

Second, for each mobile client, we generate an individual predicted value of received requests, exp_i. exp_i is created randomly, such that $exp_{total} = \sum exp_i$. In practice, this value can be obtained from the history of clients using forecasting techniques [4].

id	type	table	column	condition	value	info
18-3	slot	datebook	*	day='17-FEB-2002' AND *hour* $\geq$ 8 AND *hour* $\leq$ 13	all records that satisfy the condition	-

Figure 7: Reservations obtained in the calendar example.

Third, we generate, based on the prediction reliability factor, $sigma_{base}$, the (expected) real value of requests received in each client, dem_i. dem_i is computed using a random variable with normal distribution (with $mean = exp_i$ and $sigma = sigma_{base} \times exp_i$). We evaluate two scenarios: one with good predictions (*good*) and one with bad predictions (*bad*). In table 2, we show, for each scenario, the average difference between the value of the requests created in our experiments and the predicted values (exp_i).

Finally, during each experiment, the submission of each request is controlled by a random variable with exponential distribution, as usual. The inter-arrival rate is equal to $t \times req_{avg}/dem_i$, with req_{avg} the average quantity of each request.

Our experiments evaluate two strategies to obtain reservations. In both, mobile devices obtain reservations that expire only in the end of the experiment. In the first, *stat X*, mobile devices obtain reservations only once, in the beginning of the experiment. X is the percentage of the stock the server reserves for itself (i.e., that clients cannot reserve). The remaining is reserved by mobile devices, proportionally to exp_i (in combination with value-use reservations). The second strategy, *dyn X*, extends the first by allowing mobile devices to request additional reservations when they cannot guarantee an order with the average order quantity. A request for additional reservations is serviced using the unreserved stock. To each user, the server concedes reservations proportional to the reservations he has obtained before (so that all mobile devices can get additional reservations).

In our experiments, reservations are always obtained from the server. This approach differs from other works [2], where multiple servers can be involved in the redistribution of escrowable resources. These approaches seem more appropriate for distributed settings where connectivity among servers is reliable and fast, and hosts do not have power restrictions that advise to disconnect them for some periods.

We have obtained the following results. First, the value of requests that can be *locally committed*, i.e., transactions that can be guaranteed independently in the mobile devices. Second, the value of requests that can be *immediately committed*, i.e., transactions that can be guaranteed either in the mobile device or synchronously contacting the server (if connectivity is available). As the maximum value (and percentage) of transactions that can succeed depends on the usage rate, we have also computed the maximum value of transactions that could be committed in a system without failures. All results are presented as a percentage of this maximum value. Note that using the reevaluation mechanism, it is always possible to reach the maximum value of requests after the reservations expire.

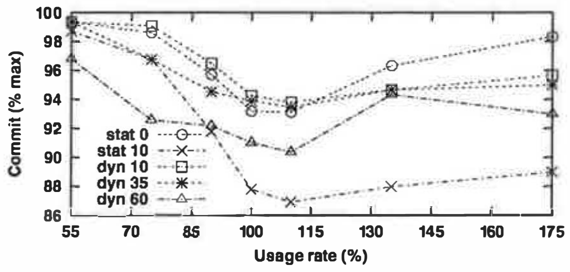


Figure 8: Locally committed transactions (good prediction).

Each experiment simulates a period of 12 hours. All results are the average of 10 runs. Different approaches are compared using the same request generation events.

6.1 Good prediction

In the first set of experiments, the difference between the actual and the expected demand is 5%, on average.

Figure 8 shows the transactions that can be locally committed. The results show that more than 85% of the maximum value of transactions can be guaranteed locally. As expected, the results are better as the usage rate deviates from 100%. For smaller usage rates, the excess of stock accommodates the unexpected requests. For bigger usage rates, as each mobile device could only obtain reservations for a fraction of the expected demand, exp_i, even if the actual demand is smaller than the expected one, all reservations tend to be consumed. Based on this rationale, we expected our results to get closer to 100% faster as usage rate deviates from 100%. Analyzing the experiments, we have discovered that the small scale of our example was introducing larger relative distortions than expected (results from the larger scale scenario confirm this conclusion – see figure 9).

Figure 10 shows the transactions that can be immediately committed. The results show that more than 95% of transactions can be immediately committed using reservations. The figure also show that a traditional client/server system (*clt/srv*) that tries to commit all transactions in the server immediately behaves much worse in this environment (where mobile devices do not have connectivity for large periods of time).

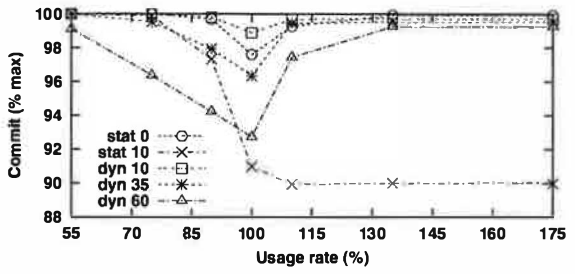


Figure 9: Locally committed transactions (good prediction, large scale scenario).

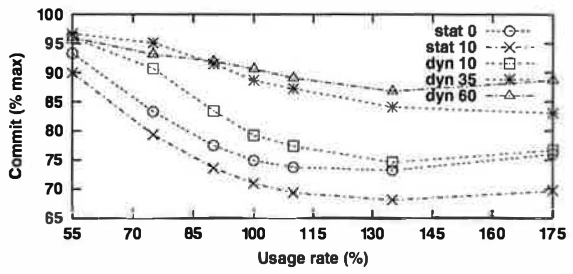


Figure 11: Locally committed transactions (bad prediction).

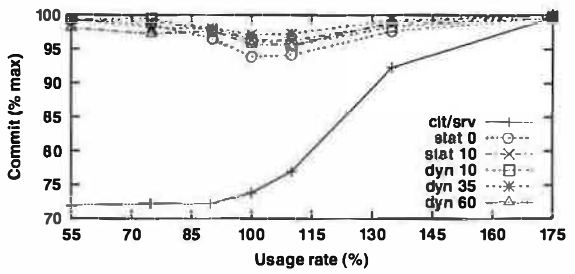


Figure 10: Immediately committed transactions (good prediction).

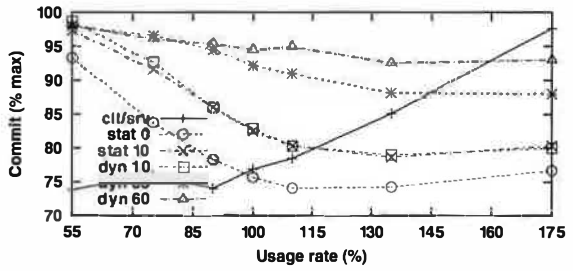


Figure 12: Immediately committed transactions (bad prediction).

6.2 Bad prediction

In the second set of experiments, we investigate if reservations can also be used successfully when predictions are bad (the difference between the actual and the predicted demand is 55%, on average).

Figure 11 shows the transactions that can be locally committed. As expected, the results are worse than the obtained when the prediction is good. In this case, obtaining reservations dynamically is much better than obtaining reservations just once. For example, in the *dyn 60* scenario, the system can commit locally more than 85% of the maximum transactions that can be committed. However, dynamic strategies requires that mobile devices communicate with the server. These communication costs increase with the usage rate (until it is close to 110%) and with the increase of the stock reserved by the server [17]. However, these costs are small – e.g. in the *dyn 60* (resp. *dyn 30*) scenario with the usage rate of 110% (resp. 90%), clients contact the server once for each 6.25 (resp. 13.6) requests locally committed (in large scale scenarios, these values are much bigger).

Figure 12 shows the transactions that can be immediately committed. The values for the dynamic strategy are disappointing as they slightly improve the results of transactions locally committed. Analyzing the experiments, we found out that the problem was due to the reservations obtained initially. To solve this problem, we have used our dynamic strategy without obtaining any reservation initially, i.e., ignoring the predicted demand (this situation is equivalent to the unavailability of predictions). The results obtained were the following.

Figure 13 shows that more than 80% of the maximum transactions can be locally committed. As the system adapts to demand dynamically, it is impossible to locally commit some of the early transactions. This explains the increase of transactions committed locally with the increase of the usage rate – when there are more transactions, the influence of these early steps tends to be smaller (results from the large scale scenario, figure 14, also corroborate this hypothesis and show a very good adaptation of the dynamic approach).

Figure 15 shows the transactions that can be immediately committed. As expected, these results are much better than those obtained when clients obtain reservations initially. In this case, more than 95% of the maximum number of transactions can be committed immediately.

The results presented show that reservations can support a mobile sales application when it is possible to estimate the demand and even when such estimation is unknown.

7 Related work

Mobile data management has been addressed in several research projects and some solutions have even been integrated in commercial products. Some of the proposed approaches are presented in [1, 16, 19].

In Oracle Lite [14], mobile clients cache database snapshots. Transactions executed in clients are integrated in the master database using the new/old write sets and detecting write/write, uniqueness and delete conflicts. Conflict resolution rules can be associated with the database

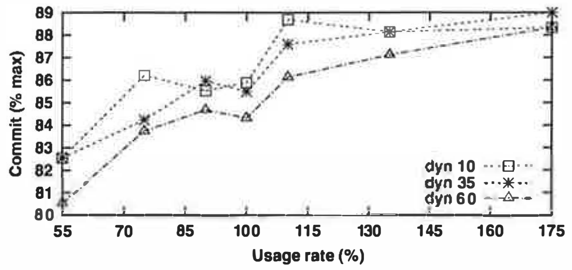


Figure 13: Locally committed transactions (bad prediction, unknown estimation).

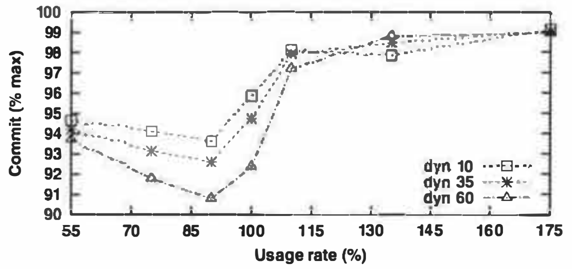


Figure 14: Locally committed transactions (bad prediction, unknown estimation, large scale scenario).

tables (and table fields). This state-based conflict resolution approach (as others [11]) is limited because the semantics of updates has been lost and it is impossible to specify any transaction-specific conflict resolution rule.

In the two-tier replication model [6], mobile nodes may propose tentative update transactions. These transactions are reapplied to the object master copy, verifying its validity using a specified acceptance rule. Invalid transactions are aborted and diagnostic messages are returned to the mobile nodes. In Bayou [20], data is replicated in a group of servers that synchronize epidemically. A primary server sets the commit order. Bayou updates allow generic conflict detection and resolution using dependency checks and merge procedures. The Mobisnap mobile transactions can be seen as an PL/SQL implementation of the updates proposed in these systems. However, unlike these systems, Mobisnap integrates mobile transactions with the reservation model to guarantee the results of transactions in the mobile devices.

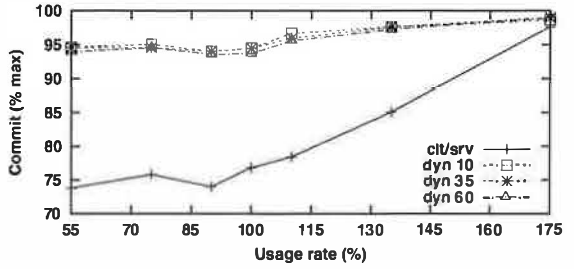


Figure 15: Immediately committed transactions (bad prediction, unknown estimation, small scale scenario).

IceCube [8] presents a reconciliation engine that tries to create an optimal single schedule, combining the maximum number of tentative actions executed in mobile devices – some action may be discarded to allow a larger set of actions to be executed. The IceCube reconciliation approach could be used in Mobisnap during the reintegration of mobile transactions received from the server and during the re-execution of transactions. In the multiversion reconciliation model [15], the server maintains multiple versions of the database history. The system tries to serialize each client transaction into one of these versions using a conflict resolution and a cost function specified with each transaction. Exploiting multiple data versions may lead to better results than the simple execution of mobile transactions in Mobisnap. However, it also imposes additional complexity in the server. Unlike Mobisnap, these systems do not include any mechanism to guarantee the results of transactions in mobile devices.

The escrow model [12] can be used to guarantee the result of some transactions in mobile devices [9]. The basic idea is to divide the available items of a commodity among several mobile sites – transactions that only use local items can be independently guaranteed. As explained, the Mobisnap escrow reservations implement and adapt this idea to an SQL-based system where reservations are used transparently. Therefore, any transaction can explore all reservations instead of only those it has been designed to. Moreover, the examples of section 4.5 show that additional reservations are necessary to guarantee the results of most transactions.

The use of escrow techniques can be generalized by exploiting object semantics [21]. The idea is to split large and complex objects into smaller fragments with certain constraints. If these constraints are honored, the modified fragments can be later merged using the semantic information available. This approach can be used only with some data types and it is more appropriate to object oriented databases.

TACT [22] defines a framework to control the divergence among several replicas integrating several metrics previously proposed. Although this approach can be used to increase the likelihood of reintegration success, it cannot guarantee the results locally. TACT demands applications to inform the system how each update affects the existent logical consistency units. This approach forces updates to know consistency units and it is error-prone, as any incorrect adjustment in any update leads to invalid divergence values. We believe that an automatic and transparent approach, as used in Mobisnap to check if mobile transactions can be guaranteed, is preferable.

A preliminary and incomplete version of Mobisnap was presented elsewhere [18]. Since then, important modifications have been introduced in the system, such as, the introduction of new reservations and the modification of the transaction processing.

8 Final remarks

The Mobisnap database middleware system is designed to support applications that run on mobile computing environments. It supports independent operation combining mobile transactions and reservations.

A mobile transaction is a small PL/SQL program submitted by an application to modify the database state. As the final result of a mobile transaction is obtained running its program in the server, it can include conflict detection and resolution rules that exploit the semantic information associated with the operations.

A reservation provides some promise upon the database state, thus guaranteeing that no conflict will arise when a mobile transaction is executed in the server. Therefore, if a mobile client holds enough reservations, it can guarantee the final result of a mobile transaction independently.

Besides implementing this conflict avoidance mechanism in a middleware SQL-based system that allows legacy clients to continue to access the database, our reservation model presents the following new contributions. First, it includes several types of reservations. The examples presented in section 4.5 show that this feature is necessary to guarantee the result of most transactions. Second, the client transparently verifies if it can guarantee the result of any mobile transaction. Besides allowing mobile transactions to be written as usual PL/SQL programs, this property allows any mobile transaction to explore all available reservations. Finally, it is integrated with mobile transactions that allow the definition of conflict resolution rules. To our knowledge, this integration makes our system unique.

This paper has focused on the description of the Mobisnap reservation model and its integration with mobile transactions. The examples presented throughout the paper exemplify the use of the model with realistic applications. The system design and implementation described demonstrate the feasibility of the model. The evaluation of the reservation model shows that reservations can be used to support independent operation in a mobile sales applications even when it is impossible to estimate the expected demand. In the future, we expect to study the use of reservations in different settings and to evaluate the impact of reservations (triggers) in the performance of the database server.

More information on the Mobisnap system can be obtained from `http://asc.di.fct.unl.pt/mobisnap`.

Acknowledgements

We would like to thank our paper shepherd, Douglas Terry, the anonymous reviewers and Marc Shapiro for their helpful comments on earlier versions of this paper.

References

[1] BARBARÁ, D. Mobile computing and databases - a survey. *Knowledge and Data Engineering 11*, 1 (1999), 108–117.

[2] CETINTEMEL, U., ÖZDEN, B., FRANKLIN, M., AND SILBERSCHATZ, A. Design and evaluation of redistribution strategies for wide-area commodity distribution. In *Proc. of the* 21st *International Conference on Distributed Computing Systems* (Apr. 2001), pp. 154–164.

[3] DAR, S., FRANKLIN, M. J., JÓNSSON, B., SRIVASTAVA, D., AND TAN, M. Semantic data caching and replacement. In *Proc. VLDB'96* (Sept. 1996), pp. 330-341.

[4] FRANK, T., AND VORNBERGER, M. Sales forecasting using neural networks. In *Proc. ICNN'97* (1997), vol. 4, pp. 2125–2128.

[5] GRAY, C., AND CHERITON, D. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. of the* 12th *ACM Symposium on Operating systems principles* (1989), pp. 202–210.

[6] GRAY, J., HELLAND, P., O'NEIL, P., AND SHASHA, D. The dangers of replication and a solution. In *Proc. of the 1996 ACM SIGMOD international conference on Management of data* (1996), pp. 173–182.

[7] GRAY, J., AND REUTER, A. *Transction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, 1993.

[8] KERMARREC, A.-M., ROWSTRON, A., SHAPIRO, M., AND DRUSCHEL, P. The icecube approach to the reconciliation of divergent replicas. In *Proc. of the* 20th *ACM Symposium on Principles of Distributed Computing* (2001), pp. 210–218.

[9] KRISHNAKUMAR, N., AND JAIN, R. Escrow techniques for mobile sales and inventory applications. *Wireless Networks 3*, 3 (1997), 235–246.

[10] KUENNING, G. H., AND POPEK, G. J. Automated hoarding for mobile computers. In *Proc. of the* 16th *ACM Symposium on Operating Systems Principles* (1997), pp. 264–275.

[11] KUMAR, P., AND SATYANARAYANAN, M. Flexible and safe resolution of file conflicts. In *Proc. USENIX Winter Technical Conference* (New Orleans, LA, USA, Jan. 1995), pp. 95–106.

[12] O'NEIL, P. E. The escrow transactional method. *ACM Transactions on Database Systems (TODS) 11*, 4 (1986), 405-430.

[13] ORACLE. Pl/sql user's guide and reference - release 8.0, June 1997.

[14] ORACLE. Oracle8i lite replication guide - release 4.0, 1999.

[15] PHATAK, S., AND BADRINATH, B. R. Multiversion reconciliation for mobile databases. In *Proc.* 15th *Int. Conference on Data Engineering* (Mar. 1999), pp. 582–589.

[16] PITOURA, E., AND SAMARAS, G. *Data Management for Mobile Computing*, vol. 10. Kluwer Academic Publishers, 1998.

[17] PREGUIÇA, N. *Data Management for collaborative mobile computing (in portuguese).* PhD thesis, Dep. Informática, FCT, Universidade Nova de Lisboa, 2003 (expected).

[18] PREGUIÇA, N., ET AL. Mobile transaction management in mobisnap. In *Proc. of ADBIS-DASFAA 2000* (2000), vol. 1884 of *Lecture Notes in Computer Science*, pp. 379–386.

[19] SAITO, Y., AND SHAPIRO, M. Replication: Optimistic approaches. Tech. Rep. HPL-2002-33, Hewlett-Packard Laboratories, Mar. 2002.

[20] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of the* 15th *ACM Symposium on Operating systems principles* (1995), pp. 172–182.

[21] WALBORN, G. D., AND CHRYSANTHIS, P. K. Supporting semantics-based transaction processing in mobile database applications. In *Proc. Symposium on Reliable Distributed Systems* (1995), pp. 31–40.

[22] YU, H., AND VAHDAT, A. Design and evaluation of a continuous consistency model for replicated services. In *Proc* 4th *Symposium on Operating System Design and Implementation (OSDI 2000)* (Oct. 2000), pp. 305–318.

Protecting Applications with Transient Authentication

Mark D. Corner and Brian D. Noble
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
{mcorner,bnoble}@umich.edu
http://mobility.eecs.umich.edu

Abstract

How does a machine know who is using it? Current systems authenticate their users infrequently, and assume the user's identity does not change. Such *persistent authentication* is inappropriate for mobile and ubiquitous systems, where associations between people and devices are fluid and unpredictable. We solve this problem with *Transient Authentication*, in which a small hardware token continuously authenticates the user's presence over a short-range, wireless link. We present the four principles underlying Transient Authentication, and describe two techniques for securing applications. Applications can be protected transparently by encrypting in-memory state when the user departs and decrypting this state when the user returns. This technique is effective, requiring just under 10 seconds to protect and restore an entire machine, but indiscriminate. Instead, applications can utilize an API for Transient Authentication, protecting only sensitive state. We describe our ports of three applications—PGP, SSH, and Mozilla—to this API. Mozilla, the most complicated application we have ported, suffers less than 4% overhead in page loads in the worst case, and in typical use can be protected in less than 250 milliseconds.

1 Introduction

How does a device know that the right person is using it? Unfortunately, authentication between people and their devices is both *infrequent* and *persistent*. Should a device fall into the wrong hands, the imposter has the full rights of the legitimate user.

Authentication requires that a user supply some proof of identity—via password, smartcard, or biometric—to a device. Unfortunately, it is infeasible to ask users to provide authentication for each request made of a device. Imagine a system that requires the user to manually compute a message authentication code [26] for each command. The authenticity of each request can be checked, but the system becomes unusable. Instead, users authenticate infrequently to devices. User authentication is assumed to hold until it is explicitly revoked, though some systems further limit its duration to hours or days. Regardless, in this model authentication is persistent.

Persistent authentication creates tension between security and usability. To maximize security, a device must constantly reauthenticate its user. To be usable, authentication must be long-lived.

We resolve this tension with a new model, called *Transient Authentication*. In this model, a user wears a small token, equipped with a short-range wireless link and modest computational resources. This token is able to authenticate constantly on the user's behalf. It also acts as a proximity cue to applications and services; if the token does not respond to an authentication request, the device can take steps to secure itself.

At first glance, Transient Authentication merely seems to shift the problem of authentication to the token. However, mobile and ubiquitous devices are not physically bound to any particular user; either they are carried or they are part of the surrounding infrastructure. As long as the token can be unobtrusively worn, it affords a greater degree of physical security.

Transient Authentication has been applied to cryptographic file systems [7] and could be extended to protect swap space [28]. These provide a good first line of defense, protecting persistent storage from physical possession attacks. If the machine has been shutdown, hibernated, or has run out of power, this is sufficient to protect the machine from attack.

Unfortunately, they do not protect applications on machines that are running or have been suspended. An application that reads data from a cryptographic file system—or receives data from a secure network connection [1, 36]—holds that data in memory without protection. Mobile devices typically suspend themselves after an idle period or in response to a user closing its lid. If the device is suspended, or running, the contents of memory may be inspected through operating system interfaces or through physically probing the memory bus. An attacker can recover passwords and sensitive data such as credit card numbers, or patient records.

One solution is to require reauthentication after suspension or an idle period. This is an insufficient solution for two reasons. First, after a suspension or timeout all sensitive, in-memory data must be flushed or protected—we are unaware of work that has addressed this problem. Second, timeouts do not address the tension in usability versus security. This paper proposes mechanisms to address both of these problems.

We first describe the trust and threat model we consider, and enumerate the four principles underlying Transient Authentication. We then present two mechanisms for protecting in-memory application state.

The first, *application-transparent protection*, provides protection within the kernel. When the user departs, all user processes are suspended and in-memory pages encrypted. When the user returns, pages are decrypted and processes restarted. Protection and recovery processes each take at most ten seconds on our hardware, and applications need not be modified to benefit from this service.

Application-transparent protection is effective but indiscriminate. There are processes that can safely continue in the user's absence, and a few processes may be able to selectively identify and protect sensitive state. Our second mechanism, *application-aware protection*, supports such applications. We provide an API for applications to use Transient Authentication services directly. We have modified three applications—PGP, SSH, and Mozilla—to make use of this API. In exchange for such modifications, these applications can be protected and restored in well under half a second, and suffer no noticeable degradation of run-time performance.

2 Trust and Threat Model

Our focus is to defend against attacks involving *physical possession* of a device or *proximity* to it. Possession enables a wide range of exploits. The easiest attack is to use authentication credentials that are cached by the operating system or individual applications. Even without cached credentials, console access admits a variety of well-known attacks; some of these result in root access. A determined attacker may even inspect the memory of a running machine using operating system interfaces or hardware probing.

Transient Authentication must also defend against observation, modification, or insertion of messages sent between mobile devices and the token. Simple attacks include eavesdropping in the hopes of obtaining sensitive information. A more sophisticated attacker might record a session between the token and laptop, and later steal a misplaced laptop in the hopes of decrypting prior traffic. We defeat these attacks through the use of well-known, secure protocols [11, 26].

Transient Authentication's security depends on the limited range of the token's radio. Repeaters or arbitrarily powerful transmitters and receivers could be used to extend this range. This is sometimes called a wormhole attack [16]. The rapid attenuation of high frequency radio signals makes attacks using powerful transmitters difficult in practice. A better solution would use timing information to detect the distance of the token from the device. This technique has been proposed by Brands and Chaum [4] and explored in the Wormhole detection project [16], though neither has built a practical implementation.

Transient Authentication does not defend against a trusted but malicious user who leaks sensitive data. It also does not consider network-based exploits to gain access to a machine, such as buffer overflow attacks [8]. Finally, we do not protect against denial of service attacks that jam the spectrum used by the laptop-token channel. Other attacks may attempt to exhaust the energy resources on the token. This can be addressed by reserving most resources on the token to deal with trusted connections [33].

The device operating system must be trusted. If the operating system has been compromised, secret information could be revealed to a third party. Protecting an operating system from modifications, such as Trojan horses, has been addressed in other work [21]. It must be assumed that if the device is stolen and used maliciously in any way, it will never again be used as trusted. Any device that has been out of the user's control for a lengthy period of time should be treated as suspect and not used.

3 Transient Authentication Principles

Transient Authentication is governed by a set of four guiding principles. First, users must hold the sole means to access sensitive resources or invoke trusted operations on the device. Second, the mechanisms to secure sensitive data do not need to be faster than people using them. Third, the system must impose no additional usability or performance burdens. Fourth, users must give explicit consent to actions performed on their behalf. This list is a refined version of principles that appeared in earlier work [25].

3.1 Tie Capabilities to Users

The ability to perform sensitive operations must ultimately reside with the user rather than her devices. For example, the keys that decrypt private data must reside on the user's token, and not on some other device.

At the same time, it is unlikely that the token—a small, embedded device—can perform large computations such as bulk decryption. Furthermore, requiring the token to perform cryptographic operations in the critical path of common actions will lead to unacceptable latency. In such cases, it may be necessary to cache capabilities on a device for performance. The results of the cryptographic operations can be cached. However, these decrypted capabilities must be destroyed when the user leaves, and the master capability cannot be exposed beyond the token.

One could instead imagine a simple token that responded to authentication challenges. This gives evidence of the user's presence but does not supply a cryptographic capability. An operating system could use this evidence to govern access to resources, data, and services. Unfortunately, this model is insufficient. If the device is *capable* of acting without the token, then an attacker with physical possession can potentially force it to do so. For example, consider memory access control. The operating system can be forced to provide the contents of physical memory through direct OS interfaces such as Linux's `/dev/mem` and Windows' `\Device\PhysicalMemory`. An encrypted memory store, with the keys stored only on the token, is not subject to the same attack.

Cached capabilities—and the data they protect—can only remain while the token is present; when the token is out of range, sensitive items must be protected. As a simple example consider a cryptographic file system. If the user leaves, an attacker could physically attack the machine, recovering the disk cache. Even if the disk is encrypted, the decryption key can be found in memory. Instead, the disk cache must be protected and the keys flushed from the system.

3.2 Secure Just Faster than People

The securing process must happen before an attacker gains access to the machine. One might think that this must happen quickly. However, since people are slow, the limit is on the order of seconds, not milliseconds.

Suppose that a malicious individual wishes to compromise a device. After stealing the device, he must take advantage of persistent authentication information. For example, a user logs in and leaves a laptop, an attacker can take the device and prevent the machine from protecting itself, reading the contents of memory at his leisure. The amount of time required for such a physical attack depends on a variety of human factors,

Some optimizations in the securing process can be made to ensure that recovery is fast enough. Rather than simply erasing sensitive information during the securing process, one might prefer to encrypt and retain it. This additional work can save time on restoration: when the user returns, the laptop can obtain the proper key from the token and decrypt the data in place, restoring the machine to pre-departure state. As long as the additional work to secure the machine is within tens-of-seconds, this is an acceptable tradeoff.

3.3 Do No Harm

Investing capabilities with users increases the security of the system. However, increases in security cannot impose additional burdens. When faced with inconvenience, however small, users are quick to disable or work around security mechanisms. Both performance and usability must remain unaffected.

Users already accept infrequent tasks required for security. For instance, passwords are used occasionally, usually on the order of once a day. More frequent requests for passwords are perceived as burdensome; a transparent authentication system can impose no usability constraints beyond those of current systems.

Transient Authentication must also preserve performance, despite the additional computation increased security requires. As long as this computation is imperceptible to the user, it is an acceptable burden. For example, the Secure Socket Layer (SSL) [14] protocol requires processing time for encryption and authentication. However, this cost is masked by network latency.

When the user returns after being away, the device must return itself to the pre-departure state. This includes user visible state such as open windows, and network connections, as well as pre-departure performance. If information was flushed, or protected using encryption, it must not take a visible mount of time to recover. Users who are forced to wait for recovery to finish are less likely to use the system.

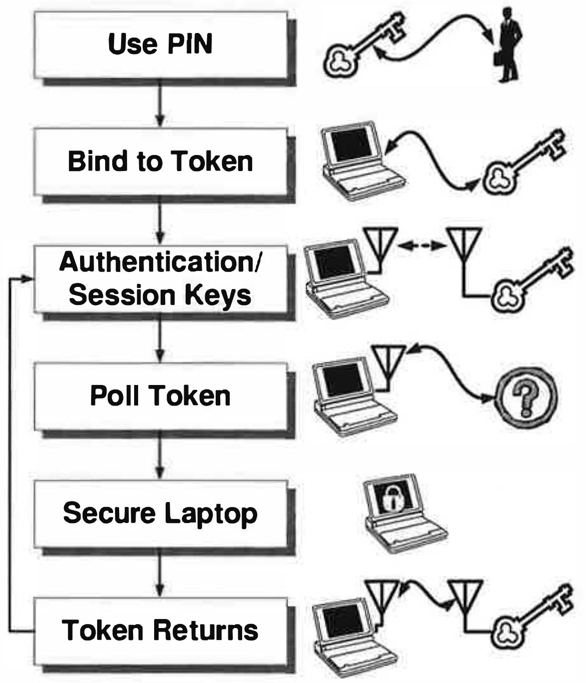
3.4 Ensure Explicit Consent

Tokens and devices must interact securely, and with the user's knowledge. In a wireless environment, it is particularly dangerous to carry a token that could provide capabilities to unknown devices autonomously. A "tailgating" attacker could force another user's token to provide capabilities, nullifying the security of the system. Instead, the user must authorize individual requests from devices or create trust agreements between individual devices and the token.

Theoretically, users could confirm every capability requested by the device. However, usability is paramount, so the granularity of authorization must be much larger. Instead of an action-by-action basis, user consent is given periodically on a device-by-device basis.

To ensure explicit consent, our model provides for the *binding* of tokens to devices. Binding is a many-to-many relationship; One might interact with any number of devices, and any number of users might share a device. Binding requires the user's assent but can be long-lived, limiting the usability burden. The binding process requires mutual authentication between device and token.

Unfortunately it is possible for a user to lose a token. Token loss is a serious threat, as tokens hold authenticating material; anyone holding a token can act as that user. To guard against this, users must periodically authenticate to the token. This authentication can be persistent, on the order of many days. This return to an un-



This figure shows the process for authenticating and interacting with the token. Once an unlocked token is bound to a device, it negotiates session keys and can detect the departure of the token.

Figure 1: Token Authentication System

bound state is similar to what Ross and Stejano call reverse metempsychosis [34]. Nominally, any authenticating material in the token is encrypted by a user-supplied password. When the authentication period expires, the token flushes any decrypted material, and will no longer be able to authenticate on the user's behalf. Placing authentication material in PIN-protected, tamper-resistant hardware [35] further strengthens the token in the event of loss or theft. The Transient Authentication process, illustrating all of these mechanisms, is shown in Figure 1.

4 Application-Transparent Protection

Applications store sensitive information, such as credit card numbers and passwords, in their virtual address space. Even with an encrypted file system [7] and swap space [28], the in-memory portions of an application's address space vulnerable to attack. The memory bus or chips may be probed by a knowledgeable attacker, or OS interfaces can be exploited to examine raw memory contents. This section describes a technique, called application-transparent protection, for protecting in-memory process state. The main benefit of this technique is that it protects processes *without* modification. The application designer does not need to identify which data structures contain secret data and users do not have to designate which processes to protect.

4.1 Design

Applying the first stated goal of Transient Authentication requires that the capability of reading memory be tied to the user. One approach would be to require each load and store to use encryption, using keys only available on the token. The performance of the machine would suffer greatly, clearly violating the principle of "do no harm". An alternative would be to protect the machine by flushing the contents of memory into the swap space and zeroing the memory whenever the user departs. This scheme would make use of swap space encryption, combined with keys available only on the token. On return, the paged-out memory would be read back from the disk into the memory pages. Unfortunately, both protecting and maintaining the machine would require a significant amount of overhead in disk operations, leaving the machine vulnerable longer and burdening the user. This would violate the principles of "securing just faster than people" and "do no harm".

Instead, the system must encrypt the virtual memory of processes in place. Since all the encryption operations are done in memory, this mechanism provides both fast protection and recovery. To avoid corrupting the encrypted memory, processes must first be placed in a hibernation state, preventing them from executing while the user is away. Certain processes can be designated as unprotected, but most processes will not execute until the user returns. On recovery, the memory is decrypted and the process is re-animated; to a returning user it appears as if nothing has changed.

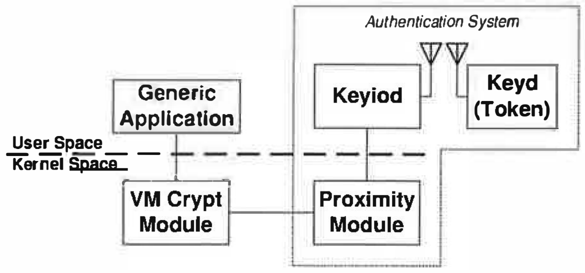
We have found that the recovery process is fast enough to remain unnoticed by users. However, if the ratio of memory size to processing speed were much greater than on our test machine, the securing or recovery process may be too lengthy. In this case, the application-aware techniques presented later in the paper will be required. If recovery is the bottleneck, it is possible to first recover applications the user will interact with quickly. Operating systems already track interactive jobs to provide good response time in process scheduling [32], enabling informed selection of recovery order. However, we expect that the current memory/processor balance will continue for the foreseeable future making this technique unnecessary.

4.2 Implementation

We have built a Linux prototype to protect the in-memory portions of application state. At startup, an in-kernel module receives a fresh key from the token to govern the memory of running processes. The module receives notifications of the token's status from another in-kernel module. When it receives notification of user departure, each processes is set to hibernate, using techniques borrowed from the Linux Software Suspend project [6]. First, each process is marked as hibernating and also as having a pending signal. The only processes allowed to continue running are essential tasks related to Transient Authentication and the operating system. The marked processes are woken up and the kernel signal dispatcher prevents the process from running until the hibernate flag is cleared. This ensures that every process is in a known hibernating state and cannot change its own memory. This has the property of having to wait for uninterruptible processes to become interruptible. However processes normally last in this state only for a short time. It may be the case that a buggy process has become stuck in an uninterruptible state; we are currently unable to handle this situation. Other hibernation methods may exist; we are looking at improving this mechanism.

After hibernation is complete, the module walks the virtual memory space of each process, looking for in-memory pages. Each in-memory page is encrypted using the pre-fetched key, and marked as such to prevent multiple encryptions of shared memory pages. The decrypted copy of the key is then thrown away. On user return, the process is reversed—the kernel fetches a decrypted version of the key from the token, the memory is decrypted and all processes are awoken from where they left off.

Free memory pages present a special difficulty. Applications may have allocated memory, stored secret information in that space, and then terminated. This memory is returned to the OS, and it may still contain remnants of that information. To protect these remnants, the module must walk the list of free pages and zero the memory of each page in the list.



This figure shows the components in the transparent protection system. When authentication is lost, a kernel module encrypts the in-memory state of any generic application. Authentication and token communication are handled by a kernel proximity module and a user space daemon.

Figure 2: Transparent Protection

An overview of the transparent protection system is shown in Figure 2. Fetching the encryption key from the token is handled by a pair of user space daemons, `keyiod` on the laptop and `keyd` on the token, communicating via a wireless link. Both `keyiod` and `keyd` are written in C, and `keyd` is compiled for the StrongARM processor used by the iPAQ. The laptop client, `keyiod`, is multithreaded to allow multiple outstanding requests, decreasing the latency of multiple requests. The token process, `keyd` is primarily compute bound and does not require acknowledgements, permitting a single threaded design.

Exposure of the virtual memory encryption key would nullify its protections, so each message between `keyiod` and `keyd` must be encrypted. Further, since the token is used to create fresh encryption keys, the link must be authenticated as well.

The kernel module, `tadev` also exports an interface for other OS services to utilize the token. The `tadev` module provides three functions: `sendmessage`, `addhandler`, and `removehandler`. This allows generic modules to send messages as well as register, and unregister to receive messages of requested types. Modules can send requests for capabilities, receive responses, as well as be informed of events such as loss of authentication.

Mutual authentication can be provided with public-key cryptography [23]. In public-key systems, each principal has a pair of keys, one public and one secret. To be secure, each principal's public key must be certified, so that it is known to belong to that principal. Because laptops and tokens fall under the same administrative domain, that domain is also responsible for certifying public keys. `Keyiod` and `keyd` use the Station-to-Station protocol [11], which combines public-key authentication and Diffie-Hellman key exchange.

Each message includes a *nonce*, a number that uniquely identifies a packet within each session to prevent replay attacks [5]. In addition, the session key is used to compute a *message authentication code*, verifying that a received packet was neither sent nor modified by some malicious third party [26].

The kernel cryptographic module must be informed when the token is no longer present. To provide this notification, we add a periodic challenge/response between the laptop and the token. These proximity polling messages are generated by a second module in the kernel. We currently set the interval to be one second; this is long enough to produce no measurable load, but adds little to the amount of time needed to protect the laptop.

5 Application-Aware Protection

Transparent application support is an effective technique, but an indiscriminate one. There are several disadvantages in protecting every process on the machine, regardless of the sensitivity of their contents. A process that only occasionally conducts sensitive operations must be completely stopped, regardless of its current tasks. Certain processes could be statically designated as non-sensitive, or the process could mark itself as sensitive dynamically. However, if two processes communicate through shared memory, both must be stopped, even though only one may be sensitive. Also, some applications that depend on constant input or network traffic may not survive the hibernation process. This burdens the user, who must either restart those applications or perform work to restore the previous state.

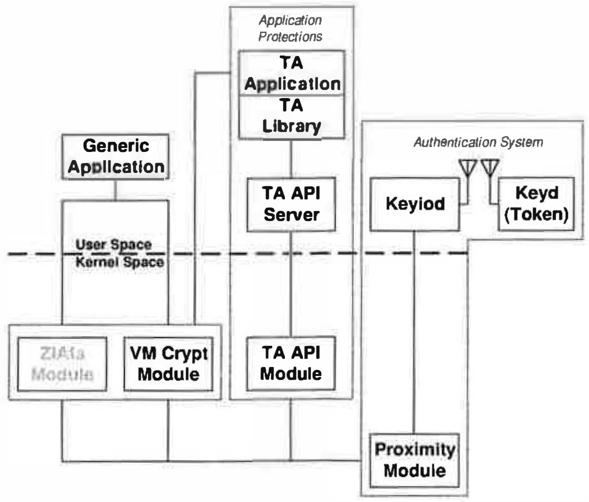
To combat these shortcomings we provide an interface for an application to manage its own sensitive information. This allows greater flexibility in handling loss of authentication and permits the application to continue to run regardless of authentication state. In order to provide this capability, we have designed an application programming interface, or API, that allows applications to use Transient Authentication services. Applications must be restructured to depend on capabilities, such as keys, held by the token. For performance, these capabilities can be cached, but they must be flushed when the token leaves.

Some applications and services already manage authentication and access to sensitive resources. Most of these systems revoke access through either explicit user logout or expiration of a long-lived session. Some of these applications and servers also provide various levels of service, depending on the specific credentials of the user. Such applications already manage identity and privilege, and would benefit from direct use of Transient Authentication services.

An overview of the system is shown in Figure 3. Generic applications can take advantage of Transient Authentication using transparent protection. Modified applications are compiled with a Transient Authentication library and communicate with the kernel using a user-space server. All interactions with the token pass through the proximity polling module and a user-space communication daemon. We have implemented parts of the system in the kernel to make the system fail-safe. If any part of the system fails, the application should still receive a notice of authentication loss.

5.1 Protecting Targeted Secrets

Identifying secret data is the most difficult part of protecting an application. The application designer must first consider the threat model and user requirements. For instance: Is all of the user's data secret? What about the meta-data? What about data received from the network? For example, the text of a word processor document is probably private, the formating of that document may or may not be, and the state of local program variables is probably not. There are no hard rules for determining these classifications and it must be left to the designer of the application.



This figure shows the various components used in the Transient Authentication system. Generic applications can be protected by the virtual memory encryption system and the ZIA file system. Modified applications are compiled with a Transient Authentication library and communicate with the kernel through a user-space server. All communications with the token go through a proximity polling module and a user-space communication daemon.

Figure 3: TA Components

Once secrets have been identified, we use two different mechanisms to tie capabilities to the token. The first is to detect when the user leaves, then encrypt secrets and forget the local copy of the key. When the user returns, that key can be retrieved from the token and the secret decrypted. The second is to always store the information encrypted, and decrypt it for every short term use.

Choosing which mechanism to use depends on the properties of the data, including size and frequency of use. Accessing and restoring secrets must not take a noticeable amount of time, and protection must be done "just faster than people". In some cases, both of the mechanisms conform to the principles of Transient Authentication, allowing the programmer to pick the more convenient option.

5.2 Application Programming Interface

Before a user starts an application that employs the Transient Authentication API, that user must have one or more *master keys* for that application installed on their token. In our implementation, master keys are 128-bit AES [9] keys. These keys must be installed by an administrative authority, and can never be exposed beyond the token. As we will see, the master key is typically used as a *key-encrypting key*, but can sometimes protect small data items directly. Once a key is installed, the

```
/* Register an application with the library */
int   ta_application_reg   ( IN char* app_name,
                             IN char* username);

typedef
enum ta_change{TA_LOSS, TA_GAIN} ta_change_t;

typedef
int (* ta_auth_hdlr_t ) ( IN ta_change_t change,
                          IN int flags );

/* Register a handler for change in
   authentication */
int   ta_auth_change_reg   ( IN int appid,
                             IN ta_auth_hdlr_t  hdlr );

typedef char* ta_keyname_t;

/* Decrypt a buffer on the token with a key */
int   ta_decr_buf  ( IN int appid,
                     IN ta_keyname_t keyid,
                     IN char* inbuf,
                     IN size_t  inlen,
                     OUT char** outbuf,
                     OUT size_t* outlen  );

/* Encrypt a buffer on the token with a key */
int   ta_encr_buf  ( IN int appid,
                     IN ta_keyname_t keyid,
                     IN char* inbuf,
                     IN size_t  inlen,
                     OUT char** outbuf,
                     OUT size_t* outlen  );
```

This listing shows the API for Transient Authentication. Three types of functions are included: registration with the user-space server, registration of authentication call-back functions, and buffer decryption using the token and previously registered key.

Figure 4: Transient Authentication API

API is available. It is summarized in Figure 4.

On startup, each protected application *registers* itself with the API, providing the its name and the user running it. We chose usernames to provide flexibility in token identities. There is no reason why this username cannot be a UID, or some other identity. The application then installs a *handler*. The handler is called when the token fails to respond to a request, revoking authentication, or when a departed token once again is in range, reestablishing authentication.

Each master key acts as the capability to perform sensitive actions on behalf of its user and application. Simple examples of such actions are reading cached passwords or credit card numbers. These items are small; it is feasible to ship encrypted copies of them to the token, decrypt them, and send them back. This can be done directly with `ta_encr_buf` and `ta_decr_buf`. The application may decrypt and cache such items, but must clear them when notified of token departure.

As implemented, the token contains a separate master key for each application. However, this is flexible and individual master keys could cover multiple applications, although this sacrifices the key isolation provided by separate keys. Although master keys are never exposed outside the token, bugs in applications could lead to possible attacks on the master keys. Also, these master keys should be escrowed by an administrative authority if they are used for any persistent, non-recoverable data. A similar escrow policy was proposed in the ZIA file system [7].

Some things cannot be handled with direct encryption and decryption. Passing large data elements directly to the token for decryption would likely impose too great of a performance penalty. To protect large elements, the application must first create a *sub-master key*. Sub-master keys cover large objects. Encrypted copies of the sub-master can be stored at any time, while decrypted copies can be kept only while authentication holds. Our idiom for creating sub-master keys is to choose a random number as the encrypted key, and have the token "decrypt" it. Although the system needs to check for resulting weak keys, this ensures that a secret key is never generated without the token's involvement.

On startup, applications do not hold any sensitive data; they must first either decrypt an item or obtain a derived key. These decryption requests will fail if the token is out of range, leaving the application in a safe state. Once the first item or key is successfully decrypted, the user is considered authenticated. Thereafter, the run time system tracks the token's comings and goings, reporting them to registered handler. The next three sections describe how we modified three user applications to use the API.

5.3 Pretty Good Privacy (PGP)

Pretty Good Privacy [1], or PGP, uses the RSA asymmetric encryption algorithm to digitally sign and encrypt data. Users possess a pair of keys, one public and one private. Data can be encrypted using the public key and only someone who knows the private key can decrypt it. The private key can also be used to sign the message, and anyone can verify the signature using the public key. PGP can be used to provide data integrity and privacy to a great variety of applications, however we will focus on email.

The most valuable secret held by PGP is the user's private key K_p. Commonly, K_p is protected by a user's password, P, denoted as $P\{K_p\}$. When using an email client, such as Pine, the user is prompted for the password on each signature or decryption operation. In adding Transient Authentication services to PGP we have chosen to preserve the original semantics of the application and minimize modifications. To do this we have protected K_p with a random password, P, encrypted by a key on the token, K_{PGP}. This chain of keys is written as $K_{PGP}\{P\}, P\{K_p\}$. The modifications made to PGP are summarized in Figure 5.

When a user asks PGP to decrypt or sign a piece of email, the private key, K_p, is required. PGP reads both $K_{PGP}\{P\}$ and $P\{K_p\}$ from the user's PGP key directory. It sends a decryption request to the token containing $K_{PGP}\{P\}$ and the token returns P. P is used to decrypt K_p and is then thrown away. The signing or decryption process uses K_p for as long as the operation takes, and the token is no longer needed.

Email encryption and decryption is a short process. To keep the modifications to PGP as simple as possible, any loss of authentication while using the private key causes the process to exit. Any secrets contained in freed memory can be protected by the zeroing of free pages in the transparent protection kernel.

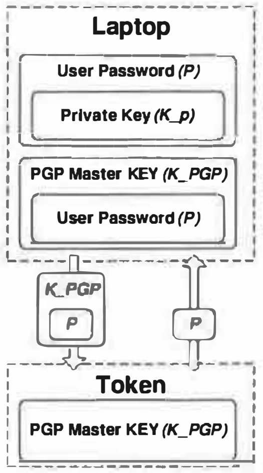
A mail program, such as `pine`, must employ PGP's output with care. For instance, if decrypted messages are displayed to the screen, the mailer must take steps to obscure that data upon loss of authentication. One possible mechanism would be to reset the display to the message index. Another option would be to redisplay the encrypted form of the message and recover the decrypted version when the user returns.

5.4 OpenSSH

The Secure Shell [36] suite of tools provides authenticated and encrypted equivalents for `rsh` and `rcp`, called `ssh` and `scp`. Client applications authenticate servers based on public key cryptography. Servers authenticate users based on passwords or public keys. Data transmitted during the session is encrypted using a key exchanged in the authentication stage. We have modified an open-source secure shell, OpenSSH; a summary of the modifications is shown in Figure 6.

OpenSSH contains two secrets that need protection, the private key, K_p, used for authentication, and the session key, S, used to encrypt data. The private key is covered by the same methods as PGP—the password, P, for K_p can be decrypted by the token's K_{SSH}.

The authentication phase generates the session key, S, which is cached. Before the session continues, OpenSSH must protect the session key. First, OpenSSH creates a new "encrypted" key, $K_{SSH}\{K_s\}$. It then uses the token to decrypt the encrypted key, yielding K_s. Finally, OpenSSH uses K_s to create an encrypted version



This figure illustrates the modifications made to PGP. The private key, K_p of the user is protected by a password, P. P is encrypted by K_{PGP}, which is only known to the token. Each time PGP needs to use K_p it asks the token to decrypt P, which enables the laptop to decrypt K_p.

Figure 5: PGP Modifications

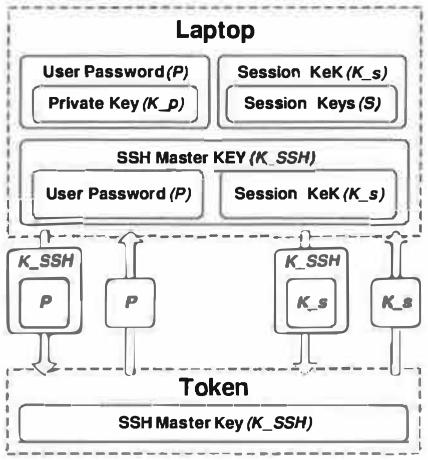
of the session key, denoted $K_s\{S\}$, which it caches.

While the user remains present, S remains decrypted in memory for session encryption and decryption. If a disconnection notification is received, OpenSSH flushes both S and K_s, but retains $K_s\{S\}$ and $K_{SSH}\{K_s\}$. When the user returns, OpenSSH must decrypt K_s using the token. It can then decrypt S and continue the session.

Each use of the session key requires a simple check that S is still available. This check takes a small amount of time, slowing data transmission by a negligible amount. If S is encrypted, the transmission of data blocks, and received data is held in the network buffer—still encrypted—until the user returns. Any blocked sessions are resumed where they left off. It may be possible for unencrypted data to get passed between the terminal and SSH after a disconnection. We are currently working on methods to prevent this from happening, such as locking the keyboard first, rejecting all data from the terminal, or returning an error to the pipe.

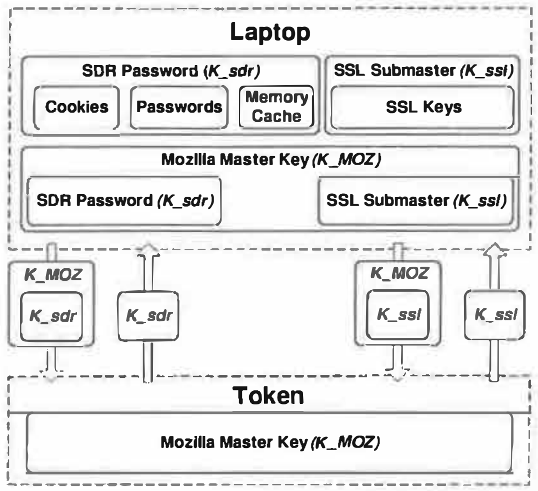
5.5 Mozilla Web Browser

Web browsers provide secure access to online accounts, e-commerce, and web-based email. Consider a typical session for accessing a secure web server at a bank. First, the browser creates a Secure Socket Layer (SSL) session with the bank's server. SSL provides session encryption to an authenticated server. The user authenticates himself by typing an account number and password into a web form. The browser often caches this information to make future logins easier. The server then



This figure illustrates the modifications made to OpenSSH. The user's private authentication key is protected by a password P, which is encrypted by a key K_{SSH}. When the user is not present, the session keys, S, are encrypted by a session key encrypting key K_s, which is encrypted by K_{SSH}, as well. When OpenSSH needs to authenticate, it uses the token to decrypt P, giving it access to K_p. Similarly, when the user returns, the token is used to decrypt K_s, giving access to the session keys.

Figure 6: OpenSSH Modifications



This figure depicts the modifications made to the Mozilla web browser. Cookies, passwords and the memory cache, all depend on Mozilla's Secret Decoder Ring for encryption and decryption. The password for the ring can be accessed using the token. SSL operates in the same way, and the sub-master key used to encrypt SSL keys can be obtained using the token.

Figure 7: Mozilla Modifications

sets a cookie on the user's local machine to authenticate future requests during this session. Note that SSL can provide for client authentication, but the vast majority of sites use cookies instead. Web pages, such as an account statement, can now be retrieved from the server and remain available in the browser's memory cache. This example identifies several places where secret information resides in the browser's address space: SSL session keys, cached passwords, cookies and the memory cache of the browser.

We have added Transient Authentication to the Mozilla web browser. Mozilla is a large and complex piece of software, containing more than 250MB of code and using several different programming languages. Some effort was made in the original source code to separate confidential and non-confidential data; however, this mostly pertained to secret keys themselves and not to sensitive data such as cookies and the memory cache. Mozilla also includes a module, the Secret Decoder Ring (SDR), that can be used to encrypt and decrypt arbitrary data. The SDR module depends on a user login to explicitly provide a decryption key. This provides an ideal location to add Transient Authentication to the system. SSL keys are contained in the same module as SDR, and therefore SSL uses these internal encryption functions, rather than the external interface. A diagram of the components in the modified browser is shown in Figure 7.

SSL session keys are used frequently, so it would be inefficient to decrypt them on every use. Instead they remain decrypted until a token departure; they are then encrypted in-place. SSL session keys could be flushed and recreated when the user returns, however to replicate the current semantics we keep the SSL session open.

Cached passwords are used very infrequently and can be stored on disk. In this case, it makes sense to have SDR decrypt the information each time it is used—Mozilla already has this capability. Cookies are used more frequently than stored passwords, but less than SSL keys. Thus, either method could be used. We have chosen to leave them encrypted and decrypt them using SDR on each use. The evaluation presented in Section 6.4 shows that this overhead is tolerable. The web cache is split into two parts, an in-memory cache and an on-disk cache. Mozilla's policy is to store data from SSL connections only in memory and never on disk. All non-SSL data is considered to be previously exposed on the network and is not protected, although there is nothing that precludes protecting this via file system encryption. The items in the memory cache are potentially large in size and frequently accessed. However, the memory cache is of limited size and can be encrypted in bulk very quickly. Thus to protect the cache, each item in the memory cache retrieved from SSL connections is SDR-encrypted on user departure and decrypted on user return. The password for SDR is erased when the user leaves and retrieved from the token when the user returns.

5.6 Application-Aware Limitations

After making the modifications to these applications we have noted several limitations. First, sensitive data may no longer be reachable in the application. These include secrets contained in leaked memory, due to programming errors in the application, and memory that has been freed. It is not possible to protect the former. However, using a pre-loaded library, calls to `realloc`, `free`, and `delete` can be intercepted and modified to zero freed memory.

Second, if the application has written secret information to the screen in a readable form, the application itself must directly obscure it; it can overwrite the data with blank pixels or other non-protected information. More generally, any secrets that have been passed to other processes may not be protected if they do not employ the API as well. We are currently looking at adding application-aware support to windowing systems, window managers, and interface toolkits.

The third difficulty is the most challenging: identifying the secrets in the application. In the examples, we have made an effort at identifying data structures containing secret data. However, this is an ongoing process that improves as we learn more about the structure of these programs. Since the modifications were not made by the original author of the applications, the effort is possibly more error-prone. In particular, if the application has made a copy of secret data that was not noticed during our examination, it will not be protected. We are currently looking at methods to analyze the flow of secrets in the memory space. One possibility may be to use language support [20].

6 Evaluation

In evaluating Transient Authentication, we set out to answer the following questions:

- What overhead does Transient Authentication impose on the system?
- Can Transient Authentication secure applications quickly enough to prevent attacks when the user departs?
- Can Transient Authentication recover application state before a returning user resumes work?

To answer these questions, we subjected our prototype to a variety of benchmarks. For these experiments, the client machine was an IBM ThinkPad X24, with 256 MB of physical memory and a 1.1 GHz Pentium III CPU. The token was a Compaq iPAQ 3870 with 64MB of RAM. They were connected by a Bluetooth [19] wireless network running in PAN mode. All encryption, except the authentication phase, is done using AES [9] with 128 bit keys. The token is somewhat more powerful and larger than current wearable devices. However, the rapid advancements in embedded, low-power devices makes this a realistic token in the near future. One possibility would be to use the IBM Linux watch [22],

	Time, sec	Over Normal
Normal (small)	0.02 (0.00)	-
With TA (small)	0.11 (0.01)	437%
Normal (large)	20.02 (1.24)	-
With TA (large)	20.06 (0.59)	0.21%

Table 1: PGP Signing and Encrypting

6.1 Transparent Protection

Transparent protection has no effect on system performance while the user is present. To measure the cost of protection and recovery we allocated 200MB of memory to a user process, occupying all available physical memory. The machine was also running a standard set of user processes, including a window manager and several shells—a total of 38 user processes not including those used for Transient Authentication and Bluetooth. We disconnected the token and reconnected it and measured the time it took to secure and recover the machine. Securing the machine required 632 microseconds to freeze all the processes, 8.92 seconds to encrypt 215.9MB of in-memory state, and 6.00 milliseconds to zero 2.25MB of free pages. On recovery, the system required 7.72 seconds to decrypt the same 215.9MB of state, and 21.2 milliseconds to unfreeze the processes. Thus, the system can encrypt state at 24MB/s, zero pages at 375 MB/s, and decrypt state at 28 MB/s. In total the machine can secure and recover our machine in less that 10 seconds each.

6.2 PGP

We subjected PGP to 50 trials of signing and encrypting two files, one 10kB in size and one 10MB. This is to simulate the two common cases of encrypting small email and large messages containing attachments. The mean and standard deviation for each experiment are reported in Table 1.

Recall that Transient Authentication-enabled PGP uses the token only for initial authentication. Therefore the only impact on performance is the additional overhead of using the token to decrypt the private key password. Both large and small files only require a small amount of overhead, although the effects are exaggerated for the otherwise fast operations on short files. In either case, the user is unlikely to notice the difference.

6.3 OpenSSH

The modified OpenSSH uses Transient Authentication for initial authentication and for protection of the session key. To measure the impact on a user's session we used a script to provide a typical user input to an `ssh` session. The script logs into another machine and runs a series of user commands: `pine`, opening a mailbox and a single message, `ls` of the home directory, running `emacs`, a `find` on a small directory, and `logout`. Between each user input there is an additional think time of two seconds. The cost of acquiring the key to login accounts for the majority of the overhead in the typical case. To measure this, we ran a second experiment: logging into a remote machine 20 times and computed the average overhead. A third experiment measures the overhead of checking for a decrypted session key on each key by using `scp` to copy a 10 MB file across the network for 20 trials. The results for each of these experiments are shown in Table 2.

	Time, sec	Over Normal (%)
Normal (session)	41.01 (0.09)	-
With TA (session)	41.31 (0.15)	0.72%
Normal (login)	0.47 (0.00)	-
With TA (login)	0.72 (0.03)	52.9%
Normal (scp)	18.96 (3.88)	-
With TA (scp)	19.21 (2.74)	1.31%

Table 2: SSH Experiments

The results show that typical user sessions are almost unaffected by use of the token—any overhead is dwarfed by think-time and the length of the session. The login micro-benchmark confirms that login accounts for most of the overhead. Long sessions also mask the additional login time, shown by the statistically identical times for modified and unmodified `scp`. We also want to know how long it takes to secure and restore `ssh` session keys. Each `ssh` session has an incoming key and an outgoing key and each are recovered separately. We instrumented ten disconnections and reconnections of the token. The results show a negligible amount of time needed for protection and 130 milliseconds, with a standard deviation of 30 microseconds, for recovery. Protecting `ssh` only requires erasing the session key. Recovering the session key requires two round-trips to the token to recover the outgoing and incoming session keys. An alternate implementation could recover both session keys simultaneously, but the cost is already small enough.

6.4 Mozilla

The only overhead to Mozilla's normal operation is the use of stored password data and cookies. Each of these are encrypted and decrypted on each use. Passwords are already SDR-encrypted and decrypted by Mozilla; our version does not add any overhead to this. To benchmark the cost of cookies, we loaded three popular pages and measured the total overhead of encryption and decryption. To put these costs in context, we also report the fastest load time we observed for each page. The cookie store was cleared between each trial and the mean and standard deviation are reported in Table 3. For these pages, the additional overhead of encrypting and decrypting cookie data is small enough to be masked by page loading times.

	Overhead,sec	Load Time,sec
CNN	0.010 (.004)	3.1
Ebay	0.035 (.001)	1.7
ESPN	0.134 (.004)	3.8

Table 3: Mozilla Cookie Overhead

We also measured the amount of time required to protect and restore Mozilla when the user leaves. To measure this we connected to two secure sites, a bank and our own department's secure web server. We disconnected the token and measured the time to safety, then reconnected the token and measured the recovery time. The results for each component are shown in Table 4. We also report the amount of data in the memory cache, and the amount of data consumed by SSL keys.

	Protect, sec	Restore, sec
Memory Cache (518 kB)	0.222 (0.002)	0.222 (0.004)
SSL Keys (788 bytes)	0.003 (0.000)	0.074 (0.006)
SDR (16 byte key)	N/A	0.066 (0.005)

Table 4: Mozilla Protection and Recovery

Recall that the contents of the memory cache and the SSL keys are encrypted when the user leaves. The memory cache, stored passwords, and cookies depend on SDR for encryption support, so SDR's key must be flushed on departure and recovered on return. Flushing the key takes a negligible amount of time. SSL uses its own key for protecting the SSL keys, and must recover it when the user returns. The total time to secure and restore Mozilla is less than four tenths of a second. Compared to the amount of time between a user entering range and resuming work, this cost will not be visible.

7 Related Work

Tokens are small devices providing authentication information for the user. A user must physically possess the token to authenticate to a local or remote machine. Examples of hardware tokens include SecureID [29], USB tokens, and smartcards [2]. SecureIDs require the user to read a password from the token and type it into the device they are authenticating to. They utilize one-time passwords [17] solving the problems that traditional password systems have. USB tokens and smartcards are inserted into the device and either transfer authentication information to the machine or must remain attached for continued operation.

Unfortunately, tokens suffer from a fundamental weakness in reauthentication. The user must frequently reauthenticate, or manually logout to ensure that the device has not been stolen while authenticated, thus caching credentials. Constant reauthentication can be accomplished by attaching the token to the device, unfortunately this encourages a user to leave the token with the device, providing little protection.

Several efforts have used proximity-based hardware tokens to detect the presence, or absence, of an authorized user. Landwehr [18] proposes disabling hardware access to the keyboard and mouse of a machine when the trusted user is away. A commercial alternative, XyLoc [13], has a software-based guard on the protected machine that refuses access when the token is absent. These systems approximate Transient Authentication, but do not adhere to its first principle. The capability to act in these systems does not reside on the token; the token is merely advisory. Since the computing system is still capable of carrying out a sensitive operation, it could be forced to do so. Sensitive operations may be relegated to a secure coprocessor [12], rendering these physical attacks more difficult.

Rather than use hardware tokens, one could instead use biometrics. However, biometric authentication schemes intrude on users in two ways. The first is the false-negative rate: the chance of rejecting a valid user [27]. For face recognition, this ranges between 10% and 40%, depending on the amount of time between training and using the recognition system. For fingerprints, the false-negative rate can be as high as 44%, depending on the subject. The second intrusion stems from physical constraints. For example, a user must touch a special reader to validate his fingerprint. Such burdens encourage users to disable or work around biometric protection. A notable exception is iris recognition. It can have a low false-negative rate, and can be performed unobtrusively [24]. However, doing so requires three cameras—an expensive and bulky proposition for a laptop.

For Transient Authentication to succeed, a computing device must *forget* sensitive information, typically through encryption. Thereafter, only the token can provide the key to recover this information. Such techniques have also been applied to revocable backups [3] and secure execution of batch jobs [30], and are sometimes referred to as non-monotonic protocols [31]. It can be difficult to completely erase previously stored values, whether in memory or on disk [15]. However, given a small amount of easily erasable media one can solve this problem for a much larger, more persistent store [10].

ZIA, a cryptographic file system, uses Transient Authentication for file data protection [7]. ZIA imposes overheads of less than 10% for representative workloads, and imposes no new usability burdens. Unfortunately, ZIA does not protect data once an application has read it. Application data that is paged out can be protected [28], leaving only in-memory state vulnerable to attack.

The Resurrecting Duckling security policy [34] proposes a set of policies for binding wireless devices to an owner. Our approach is similar in that laptops and tokens are bound by a user action, and trusted until a timeout period. In the duckling parlance, the binding process is "imprinting" and the authentication timeout causes token "assassination". Bluetooth [19] uses similar techniques to "bond" two devices in a trust relationship and bonds can be removed manually.

8 Conclusion

Mobile devices are susceptible to loss or theft, leaving the state of running applications vulnerable to data exposure. Current methods of authentication do not solve this problem since authentication is both infrequent and persistent. As a solution to this problem, we propose Transient Authentication, which allows a system to constantly reaffirm the capability to read sensitive data from memory, while giving the user no reason to turn protections off.

In this paper, we have demonstrated two protection methods that use Transient Authentication support. One mechanism is transparent, operating without application modification. The second is an API that gives greater flexibility to application designers in dealing with authentication. The evaluation of these two techniques shows that transparent protection can both secure and recover the entire physical memory of the machine within 10 seconds and that the API can be used to secure a complex application within four tenths of a second.

Acknowledgements

The authors wish to thank Peter Chen and Peter Honeyman, for many valuable conversations about this work. Landon Cox, Minkyong Kim, and James Mickens provided helpful feedback on earlier drafts. We would also like to thank our anonymous reviewers and our paper shepherd Adrian Perrig for their thoughtful comments.

This work is supported in part by the Intel Corporation; Novell, Inc.; the National Science Foundation under grant CCR-0208740; and the Defense Advanced

Projects Agency (DARPA) and Air Force Materiel Command, USAF, under agreement number F30602-00-2-0508. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Intel Corporation; Novell, Inc.; the National Science Foundation; the Defense Advanced Research Projects Agency (DARPA); the Air Force Research Laboratory; or the U.S. Government.

References

[1] D. Atkins, W. Stallings, and P. Zimmermann. PGP message exchange formats. RFC 1991, August 1996.

[2] M. Blaze. Key management in an encrypting file system. In *Proceedings of the Summer 1994 USENIX Conference*, pages 27–35, Boston, MA, June 1994.

[3] D. Boneh and R. J. Lipton. A revocable backup system. In *Proceedings of the 6th USENIX Security Symposium*, pages 91–96, San Jose, CA, July 1996.

[4] S. Brands and D. Chaum. Distance-bounding protocols. In *Proceedings of EUROCRYPT '93*, Lecture Notes in Computer Science, no. 765, pages 344–359. Springer-Verlag, 1993.

[5] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[6] F. Chabaud. Linux software suspend. SourceForge.

[7] M. D. Corner and B. D. Noble. Zero-interaction authentication. In *Proceedings of the ACM International Conference on Mobile Computing and Communications*, pages 1–11, Atlanta, GA, September 2002.

[8] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer overflow attacks. In *Proceedings of the Seventh USENIX Security Symposium*, pages 63–77, San Antonio, TX, January 1998.

[9] J. Daemen and V. Rijmen. AES proposal: Rijndael. Advanced Encryption Standard Submission, 2nd version, March 1999.

[10] G. Di Crescenzo, N. Ferguson, R. Impagliazzo, M. Jakobsson, C. Meinel, and S. Tison. How to forget a secret. In *Proceedings of the 16th Annual Symposium on Theoretical Aspects in Computer Science*, pages 500–509, Trier, Germany, March 1999.

[11] W. Diffie, P. van Oorschot, and M. Wiener. *Design Codes and Cryptograhpy*. Kluwer Academic Publishers, 1992.

[12] J. Dyer, M. Lindemann, R. Perez, R. Sailer, S.W· Smith, L. van Doorn, and S. Weingart. Building the IBM 4758 secure coprocessor. *IEEE Computer*, 34(10):57–66, October 2001.

[13] Ensure Technologies, Ann Arbor, Michigan.

[14] A. Freier, P. Karlton, and P. Kocher. The SSL protocol version 3.0. Internet Draft, March 1996.

[15] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th USENIX Security Symposium*, pages 77–89, San Jose, CA, July 1996.

[16] Y. Hu, A. Perrig, and D. B. Johnson. Wormhole detection in wireless ad hoc networks. Technical report, Rice University Department of Computer Science, June 2002.

[17] L. Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–72, November 1981.

[18] C. E. Landwehr. Protecting unattended computers without software. In *Proceedings of the 13th Annual Computer Security Applications Conference*, pages 274–283, San Diego, CA, December 1997.

[19] B. A. Miller and C. Bisdikian. *Bluetooth revealed.* Prentice Hall, Upper Saddle River, NJ, 2001.

[20] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[21] W· A. Arbaugh N. Itoi, J. Pollack S, and D. M. Reeves. Personal secure booting. In *Proceedings of ACISP 2001*, Syndney, Australia, July 2001.

[22] C. Narayanaswami and M. T. Raghunath. Application design for a smart watch with a high resolution display. In *Proceedings of the Fourth International Symposium on Wearable Computers*, pages 7–14, Atlanta, GA, October 2000.

[23] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–9, December 1978.

[24] M. Negin, T. A. Chemielewski Jr., M. Salganicoff, T. A. Camus, U. M. Cahn von Seelen, P. L. Venetianer, and G. G. Zhang. An iris biometric system for public and personal use. *IEEE Computer*, 33(2):70–5, February 2000.

[25] B. D. Noble and M. D. Corner. The case for transient authentication. In *Proceedings of the 10th ACM SIGOPS European Workshop*, Saint-Emillion, France, September 2002.

[26] National Institute of Standards and Technology. Computer data authentication. FIPS Publication #113, May 1985.

[27] P. J. Phillips, A. Martin, C. L. Wilson, and M. Przybocki. An introduction to evaluating biometric systems. *IEEE Computer*, 33(2):56–63, February 2000.

[28] N. Provos. Encrypting virtual memory. In *Proceedings of the Ninth USENIX Security Symposium*, pages 35–44, Denver, CO, August 2000.

[29] RSA Security, Bedford, Massachusetts.

[30] A. D. Rubin and P. Honeyman. Long running jobs in an authenticated environment. In *Proceedings of the 4th USENIX Security Symposium*, pages 19–28, Santa Clara, CA, October 1993.

[31] A. D. Rubin and P. Honeyman. Nonmonotonic cryptographic protocols. In *Proceedings of the Computer Security Foundations Workshop*, pages 100–116, Franconia, NH, June 1994.

[32] A. Silberschatz and P. B. Galvin. *Operating Systems Concepts*, chapter 5: CPU Scheduling. John Wiley & Sons, Fifth edition, 1999.

[33] F. Stajano. *Security for Ubiquitous Computing.* Wiley and Sons, West Sussex, England, 2002.

[34] F. Stajano and R. Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *Security Protocols, 7th International Workshop Proceedings*, Lecture Notes in Computer Science, 1999.

[35] B. Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of the First USENIX Workship of Electronic Commerce*, pages 155–70, New York, NY, July 1995.

[36] T. Ylonen. SSH—Secure login connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, San Jose, CA, July 1996.

iFlow: Middleware-assisted Rendezvous-based Information Access for Mobile Ad Hoc Applications

Zongpeng Li, Baochun Li, Dongyan Xu, Xin Zhou *

Abstract

Due to node mobility and limitations on bandwidth availability in wireless channels, there exist unique challenges towards achieving efficient and effective information access in wireless ad hoc networks with mobile nodes. In this paper, we address two critical questions: (1) How may information be accessed with the highest degree of bandwidth efficiency? and (2) How should algorithms be designed so that node mobility contributes positively towards high performance and efficiency? We present iFlow, a middleware-based framework for bandwidth-efficient and delay-aware information access for mobile ad hoc applications. We present the case of information rendezvous, where the demands for information are satisfied by the supplies in a fully distributed fashion, across third-party nodes beyond information suppliers and consumers. Such rendezvous is achieved via controlled diffusion of information from the suppliers, matched by the gleaning process on the consumers. We validate our claims using simulation and experimental results.

1 Introduction

The driving force and technology push for next-generation wireless networks are, and will always remain to be, the *applications*. A better understanding of the needs of emerging applications and wireless services leads to better designs of network protocols. On the other hand, the behavior of applications are diverse and often unpredictable. We may need to exert influence and control over such behavior, so that their needs are better understood and, in some cases, mathematically tractable. Particularly, *information access* is ubiquitously required in most of such applications. The information flow across wireless networks may exhibit specific patterns. From the point of view of resource utilization, we may prefer the patterns that may achieve the optimal bandwidth efficiency, as long as the requirements of applications are satisfied. This is especially the case in hybrid wireless networks that include ad hoc networks, where bandwidth efficiency is critical to their operations [1].

In this paper, we seek to design a middleware framework and middleware-based algorithms to achieve *bandwidth-efficient* information access, tailored to the needs of distributed applications on mobile ad hoc networks (which we refer to as *mobile ad hoc applications*). Particularly, we consider the case where application components on a subset of the nodes are information *suppliers*, while other components may be the *consumers*. Specific scenarios include: (1) ad hoc sensor networks where a subset of nodes are "sensors" that supply environmental data, and others are "reporters" that deliver sensor data to the users [2, 3]; and (2) hybrid wireless networks with a subset of "gateway" nodes to the Internet that supply information from the web to the regular nodes [4]. Such a case of suppliers and consumers does not limit its generality: with any applications, a node may either be a supplier or a consumer (or both) at any given time, constituting a web of supplier-consumer relationships. Without loss of generality, we use the example of hybrid wireless networks with gateway nodes as an example in this paper.

In such hybrid wireless networks, we present *iFlow*[1], a middleware architecture and a set of distributed algorithms to control the behavior of information access in mobile ad hoc applications, so that the goal of maximizing bandwidth efficiency with the presence of node mobility may be achieved. We identify the advantages of *information rendezvous*, where the demands for information are satisfied by the supplies in a fully distributed fashion, across third-party peer nodes beyond information suppliers and consumers. Such rendezvous is achieved via *controlled diffusion* of information from the suppliers, matched by the *gleaning* process on the consumers. In other words, requests are satisfied by results on third-party nodes in between suppliers and consumers. Beyond information rendezvous, we propose to

*Zongpeng Li, Baochun Li and Xin Zhou are affiliated with the Department of Electrical and Computer Engineering, University of Toronto. Their email addresses are {*arcane,bli,zhoux*}@*iqua.ece.toronto.edu*. Dongyan Xu is affiliated with Department of Computer Sciences, Purdue University. His email address is *dxu@cs.purdue.edu*. This work is sponsored in part by Bell Canada University Laboratories.

[1] *iFlow* stands for *information flow*. Our goal is the efficient flow of information across the network in mobile ad hoc applications.

1

apply network coding on third-party nodes, so that they may transmit recoded data to achieve even higher bandwidth efficiency.

The original contributions brought forth by the iFlow architecture are the following: (1) We have analyzed the case of activating controlled diffusion compared with separate information access from individual nodes, and show that in most cases iFlow contributes to achieving better bandwidth efficiency. (2) In iFlow, we have identified the relationship between the delay tolerance of applications and achievable bandwidth efficiency, so that for more delay-insensitive applications, bandwidth efficiency may be further improved. (3) Unlike some of the previous work, we explicitly consider node mobility in iFlow, and design adaptive algorithms such that the degree of node mobility contributes positively towards high performance and efficiency. (4) To further exploit available bandwidth and increase the efficiency of information access, we introduce the extensive application of *coding* in iFlow, starting from *erasure codes* (such as Tornado codes) used in information suppliers, complemented by *network coding* used in third-party peer nodes. Since both Tornado codes and network coding use efficient linear codes (e.g., the basic exclusive-or operation), the computational overhead introduced is minimal compared with the bandwidth efficiency gained with such coding processes.

In addition to analytical contributions supported by simulations, we have realized the architecture by implementing iFlow as a layer of middleware components with the Microsoft Component Object Model (COM) technology. The iFlow COM-based middleware exposes interfaces for the applications to invoke, as a wrapper around OS system calls. On the other hand, the application needs to implement event handlers to handle iFlow-specific events delivered by the middleware. Using standard Rapid Application Development tools such as Microsoft Visual Basic, customized event handlers are straightforward to implement and add to existing application functions.

The remainder of this paper is organized as follows. Sec. 2 presents the architecture and algorithms of iFlow, Sec. 3 presents the case of using network coding to further improve bandwidth efficiency. Sec. 4 presents simulation results. Sec. 5 presents a prototype implementation of the iFlow middleware framework to control mobile ad hoc applications. Finally, Sec. 6 and Sec. 7 compare iFlow with related work and conclude the paper.

2 iFlow: Algorithms and Analysis

The iFlow architecture is designed to serve as a middleware framework to support mobile ad hoc applications, whose components are distributed on different nodes in a mobile wireless ad hoc network. The iFlow architecture may be presented and analyzed from two different aspects. From the *horizontal* point of view, iFlow has included a set of fully-distributed algorithms for different application components residing on different nodes to interact with each other. Information flows from the suppliers and satisfies requests from consumers at rendezvous points, which usually are third-party peer nodes beyond the original suppliers. From the *vertical* point of view, iFlow is a middleware architecture designed to control the pattern of requesting and diffusing information in mobile ad hoc applications. In our implementation, such middleware components are implemented with Microsoft COM. Fig. 1 illustrates the iFlow architecture from both the horizontal and the vertical points of view.

In this section, we present and analyze the design of the distributed algorithms in the iFlow architecture, including the aspects of information rendezvous and source erasure codes such as Tornado codes. For the purpose of simplifying analysis and presentation of algorithms, we first focus on the *single-supplier* case, where there is a unique information supplier in the network. Based on insights and conclusions derived from the single supplier case, we then extend our discussions to include the *multiple-supplier* case, where multiple suppliers exist when the controlled diffusion process is activated.

2.1 iFlow Overview: the Single-Supplier Case

For the remainder of the paper, we consider mobile ad hoc applications deployed in a wireless ad hoc network with m mobile nodes, some of which are suppliers or consumers of a certain piece of information, referred to as a *data item*.

In the single-supplier case, we consider the availability of a unique information supplier, who resides on one of the nodes in the network, and possesses complete information of α, the data item of interest. In the example of hybrid wireless networks, such a node may be the "gateway" node to the Internet via dual network interfaces.

We propose to activate a *controlled diffusion* process so that the data item may be diffused to a subset of third-party nodes referred to as *reservoir* nodes. Each of the reservoir nodes holds a certain subset of the data item, and they collectively achieve a certain degree of *saturation* of the data item in the entire wireless network. Once a certain level of saturation is reached, a consumer of the data item that moves around within the network may use the *gleaning process* to gather the requested segments of data from neighbors who are reservoir nodes of this particular item. We argue that, for data items of moderate and high popularity, it is more bandwidth efficient to use the strategy of controlled diffusion and gleaning, rather than directly sending individual requests to the suppliers. This general idea of taking advantage of third-party peer nodes in the process of information access is referred to

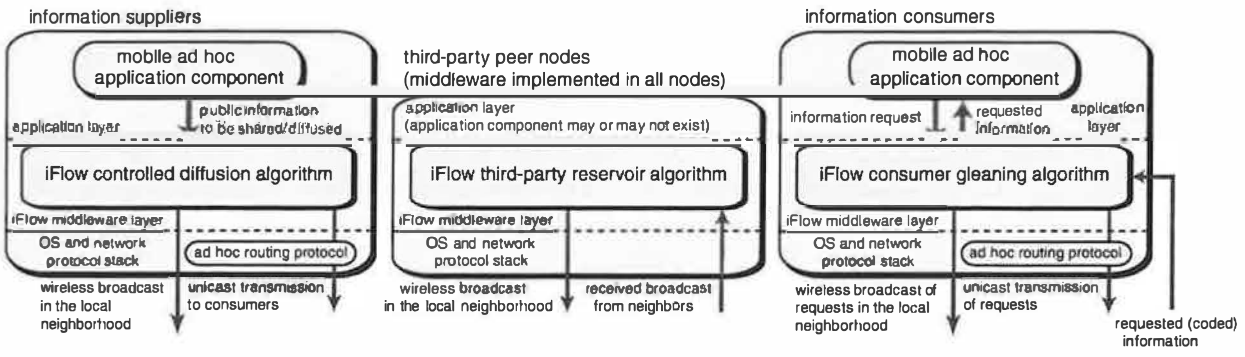

information suppliers
mobile ad hoc application component
public information to be shared/diffused
application layer
iFlow controlled diffusion algorithm
iFlow middleware layer
OS and network protocol stack
ad hoc routing protocol
wireless broadcast in the local neighborhood
unicast transmission to consumers
third-party peer nodes (middleware implemented in all nodes)
application layer (application component may or may not exist)
iFlow third-party reservoir algorithm
iFlow middleware layer
OS and network protocol stack
wireless broadcast in the local neighborhood
received broadcast from neighbors
information consumers
mobile ad hoc application component
information request
requested information
application layer
iFlow consumer gleaning algorithm
iFlow middleware layer
OS and network protocol stack
ad hoc routing protocol
wireless broadcast of requests in the local neighborhood
unicast transmission of requests
requested (coded) information

Figure 1: The iFlow architecture

as *information rendezvous*, since the requests for information are satisfied by peer nodes beyond the suppliers and consumers. The iFlow architecture is illustrated in Fig. 1.

2.2 The Controlled Diffusion Process

Assume that α is the data item of interests, and q_α is its *popularity*, *i.e.*, the percentage of nodes in the network that are consumers of α. When q_α is estimated to be beyond a threshold value, the supplier initiates the controlled diffusion process. The first step is to encode α into *coded symbols* using Tornado coding. The digital fountain approach proposed by Byers *et al.* [5] has included a detailed presentation of Tornado codes and their applications. We include a brief introduction as follows. The Tornado coding scheme generates kn coded symbols out of n uncoded data segments using the bitwise exclusive-or operation ($\oplus$). Coded symbols and uncoded segments are of equal sizes. The number k is referred to as the *stretch factor*. The coding scheme is designed such that a node that collects $n + \epsilon$ symbols is expected to be able to recover the uncoded data segments by applying substitutions and $\oplus$ operations, where $n + \epsilon$ is a number slightly larger than n. The ratio $1 + \epsilon/n$ is referred to as the *decoding inefficiency*. Well designed Tornado coding schemes may achieve decoding inefficiency that is less than 1.05. Compared to other erasure codes such as Reed-Solomon codes, Tornado codes are designed to be computationally efficient for both encoding and decoding processes.

The advantages of diffusing encoded symbols are two fold. First, it provides higher robustness for the system; second, it helps the gleaning process to complete in a timely fashion. We briefly illustrate the second point with an example below. Consider two alternative approaches of diffusing kn segments/symbols of a data item containing n data segments: (1) cyclic repetition, in which the n original data segments are diffused in order, and this procedure is repeated for k times, and (2) the n original data segments are first coded into kn encoded symbols using Tornado coding, which are then diffused into the network. For simplicity, assume that each diffused segment/symbol is held by a distinct node in the network. In the first alternative, to obtain the first segment, a consumer needs to contact one of the kn reservoir nodes; for the second through the last segments, the number of nodes that can provide A with useful segments decreases as follows: $(n-1)k, (n-2)k, \ldots, 2k, k$. Considering that k is usually a small number (e.g., 3), and the total number of nodes present in the network is usually large, the opportunity of encountering one of the k nodes is small, therefore the last few of the segments are exceedingly hard to collect. On the other hand, when erasure codes are used, the number of nodes that can provide A with useful (coded) symbols decreases in a much more graceful manner: $kn, kn-1, kn-2, \ldots, (k-1)n+1$. In this case, $(k-1)n+1$ nodes are able to provide the final symbol to A. In comparison, $(k-1)n+1$ is a much larger number than k, except for extreme cases, e.g., $n = 1$. Such extreme cases correspond to data items of very small sizes, which are not what Tornado coding targets for. The information rendezvous approach can still be applied, but with stricter requirements on data popularity and delay tolerance. In the remainder of this paper, we focus on relatively large data items for which Tornado coding can be applied to facilitate information gleaning; after all, disseminating larger data items consumes more bandwidth.

After coding the n segments in α into kn coded symbols, the supplier subsequently broadcasts these kn symbols in their original order, one after the other. The algorithm for the controlled diffusion process is shown in Table 1.

There exists a random pause between consecutive broadcasts (represented by a random variable T_x) conforming to the uniform distribution, the expected length of which, $E[T_x] = t_x$, is a parameter dependent on the degree of node mobility. Such a random pause is introduced to diversify the set of nodes covered by the diffusion pro-

Table 1 : The controlled diffusion process

```
On information supplier: controlled diffusion algorithm

consider a data item $\alpha$ on the supplier:
  if popularity $q_\alpha$ reaches threshold value
    apply Tornado coding on $\alpha$ to generate symbols of $\alpha$
    for each coded symbol $x$
      pause for a random time period $T_x$
        (a random variable), s.t. $E[T_x] = t_s$
      broadcast $x$ to neighbors
    end
  end
```

```
On reservoir nodes: third-party reservoir algorithm

Upon receiving a diffused symbol $x$:
  if $x$ is a fresh symbol not previously received
    buffer $x$
    if predefined reach has not been exceeded
      compute relay probability $p$
      if probability test on $p$ succeeds
        broadcast $x$ to neighbors
      end
    end
  end
Upon receiving a probe from a consumer:
  if able to provide requested symbols
    advertise requested symbols in possession
    if confirmation received from consumer
      transfer advertised symbols
    end
  end
```

cess. Ideally, the broadcasting node is located within a relatively different neighborhood during each individual broadcast session. This way, with the same overhead of bandwidth, the diffused symbols are distributed onto a larger number of reservoir nodes within a larger geographical area, which helps the diffused information saturate the network more uniformly. Uniform saturation is desirable in iFlow, since it eliminates the existence of "information void" — a large network area without reservoir nodes holding diffused symbols, which may lead to prolonged gleaning time for consumers residing within the area.

Note that in this paper, we use the term "broadcast" to refer to local broadcasts within the immediate neighborhood of the transmitting node, which can be accomplished using *only a single transmission* due to the local broadcast nature of wireless transmissions using omnidirectional antennas. Such an observation is sometimes referred to as the *wireless broadcast advantage* [6].

In the controlled diffusion process, each diffused symbol is accompanied by two control parameters: the *reach* of diffusion, which is the maximum number of wireless hops that a symbol may be relayed during the diffusion process by reservoir nodes, and the *relay probability* p, which is the probability that a reservoir node receiving a diffused symbol will re-broadcast the symbol. A reservoir node always buffers a fresh symbol received in the diffusion process, regardless of its decision on whether to relay that particular symbol. The reach and the relay probability are used to control the bandwidth consumption of diffusion, as well as the expected number of reservoir nodes that receive each symbol being diffused, which we refer to as the *coverage* of diffusion (denoted as c). In comparison, we define the degree of *saturation*, s, of α in a network as the ratio of the average number of symbols received by a node over the number of uncoded segments in α. For example, for a 300-node network and a diffusion process targeting 100 data segments of α, with a stretch factor of 3, 300 symbols are produced by Tornado codes. If we simply assume that, on average, 10 copies of each symbol have been buffered at reservoir nodes, 3000 symbols may then exist in the network. The average number of symbols received by each node is, therefore, 10. In this example, the degree of saturation s is $10/100 = 0.1$. The extreme case is when $s = 1$, where no gleaning process is required — all nodes may reconstruct original copies of α.

The coverage of diffusion is an increasing function of both the reach and the relay probability p. However, for the same coverage, we have the choice of using a smaller reach with larger relay probabilities, and the alternative of using a larger reach with smaller relay probabilities, as illustrated in Fig. 2. The latter approach is more desirable for two reasons. First, it introduces less overlap among different broadcasts, and is therefore more bandwidth efficient (*i.e.*, for the same bandwidth more reservoir nodes are reached). Second, it spreads symbols over a larger range of geographical area in a sparser fashion, which, aided by node mobility, is helpful to achieve uniform saturation more promptly.

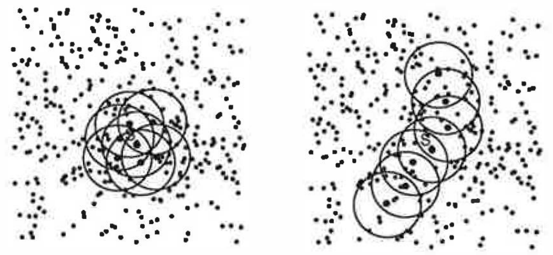

(a) Smaller reach with larger relay probabilities. (b) Larger reach with smaller relay probabilities.

Figure 2: Different choices of reach and relay probability in the controlled information diffusion process

Therefore, in the ideal scenario, we wish to employ a few of the neighboring nodes leading to different directions of the supplier to re-broadcast the diffused symbol, and each re-broadcasting reservoir node employs one successive neighbor to further relay the symbol. From the point of view of the overall diffusion process, we may observe a few non-overlapping routes extending from the supplier towards different directions, while nodes within one hop range of the routes are covered by the diffusion, and subsequently become reservoir nodes that buffer the diffused symbol.

To approximate this ideal scenario in the iFlow algorithms, we need to select nodes from the neighborhood of the supplier that are far apart from one another, so that their coverage areas overlap as slightly as possible. This objective may be achieved, if — rather than allowing each of the neighbors of the supplier to make a random and independent decision on relaying — we allow a neighbor to relay a diffused symbol if and only if it has not heard a neighbor doing exactly the same. The result of such a modified algorithm will be that, *two or three neighbors* (who are beyond the transmission range of each other) are expected to re-broadcast the symbol, and the other neighbors remain "silent".

In addition, we need to guarantee that only one neighbor of each broadcasting node further relays the symbol being diffused, if it is not beyond the predefined reach from the supplier. This may be achieved if the relay probability is set to be inversely proportional to the number of new nodes that receive the symbol during a broadcast. As shown in Fig. 3, this number can be estimated as $\rho S_\delta/(\pi R^2)$, where ρ is the average node degree in the network, and S_δ is the area covered by the downstream broadcaster B, but not by the upstream broadcaster A, which corresponds to the shaded area in the figure. The distance between A and B can be estimated as $\int_0^R r(2\pi r)\mathrm{d}r / \int_0^R 2\pi r \mathrm{d}r = 2R/3$, which is the expected distance between an arbitrary pair of neighbors. It then follows that the estimate on the number of new nodes being covered can be computed as 0.42ρ.

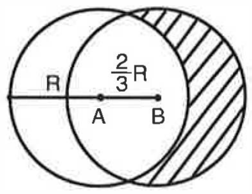


Figure 3: Effective coverage of a successive broadcast

2.3 The Information Gleaning Process

Table 2 presents the algorithm for the information gleaning process on consumers. In the gleaning process, a consumer that generates a request for data item α probes its neighbors for symbols of α as it moves around. Similar to the diffusion process, the requesting node waits for a random time period (represented by a random variable T_c) between two consecutive probes, such that its

Table 2: The information gleaning process

```
On information consumers: consumer gleaning algorithm

while not sufficient symbols to recover $\alpha$ do
    pause for random time period $T_c$
        (a random variable), $E[T_c] = \Delta t$
    broadcast a probing message to neighbors
    pause for time period $t'_c$
    if advertisements received
        confirm with node that can provide max # of symbols
        receive symbols from confirmed neighbor
    end
end
```

set of neighbors may experience some variations during that period. In our analysis, we set the expected length of this waiting period as $E[T_c] = \Delta t$, where Δt is the expected time that a node encounters one new neighbor in its neighborhood. The probe message contains a description of the symbols that the consumer already has for the requested data item α. Upon receiving a request, a neighboring node advertises the symbols it is able to provide for α, if such symbols exist. The consumer confirms with the neighbor that can provide the maximum number of symbols, after which the transfer begins. If the symbols collected after the transfer are still not sufficient to recover α, or if no advertisement is received, the consumer waits to probe again after a subsequent random time period.

To accommodate the interests of applications with stricter deadlines, we include a *panic mode* in the gleaning process. The mobile ad hoc application has the option of specifying a delay requirement for a particular request. In this case, the consumer terminates the gleaning process when the gleaning time reaches the specified delay, and enters the panic mode. In the panic mode, the consumer contacts the supplier immediately with a list of symbols it has collected. The supplier may then deliver complementary symbols to the consumer directly using a separate multi-hop unicast transmission, until the consumer has sufficient symbols to recover α.

However, since multi-hop unicast transfers incur higher bandwidth costs (which is against iFlow's objective of improving bandwidth efficiency), the panic mode should only be considered as a last resort that provides a hard delay guarantee for our rendezvous algorithms, which are inherently probabilistic. Naturally, we would like to control the diffusion process so that the network is saturated to a certain degree, where the expected gleaning time is less than the application-specified delay.

From the perspective of bandwidth efficiency, the degree of saturation is determined by the actual bandwidth consumption of the diffusion process (*i.e.,* the more band-

Table 3: List of mathematical notations

parameter	definition
m	total number of nodes within the network
ρ	average node degree of the network
Δt	expected time it takes for a node to encounter a new neighbor
α	data item of interest
n	number of uncoded data segments in the data item
k	stretch factor of Tornado coding
q_α	popularity of a data item α, *i.e.*, the percentage of nodes that eventually generate a request for α as consumers
c	coverage of diffusion, *i.e.*, expected number of reservoir nodes that hold each symbol after diffusion
s	saturation, *i.e.*, average number of symbols a node receives in diffusion over the number of uncoded data segments in the data item

width used, the higher the saturation). As a minimum requirement, the bandwidth consumption incurred by the information rendezvous process (including both diffusion and gleaning) should be (much) *less* compared with the approach of making separate unicast requests from consumers directly to the supplier. We use such a guideline as one of the design requirements of the iFlow algorithms.

We proceed to analyze critical trade-offs and relationships between two pairs of parameters: (1) the relationship between the degree of saturation and gleaning time; and (2) the relationship between bandwidth consumption and the degree of saturation, especially when compared with the all-unicast approach without using iFlow. For clarity, we list the mathematical notations of several key parameters in Table 3.

2.4 Bandwidth Consumption vs. Saturation

For this part of the analysis, we assume that the sizes of the coded symbols are much larger than the sizes of control messages in iFlow or underlying network protocols (such as routing). Therefore, we focus on the bandwidth consumption incurred when transferring the symbols across the network. More specifically, we calculate the times that the symbols are being transferred. Thanks to the wireless broadcast advantage, each local unicast or broadcast of a particular symbol counts as *a single transmission* (*i.e.*, the bandwidth consumption is 1, with a unit of symbols · hops). Further, it is straightforward to observe that, the total number of symbols replicated in the controlled diffusion process is $c \cdot kn$. Therefore, the degree of saturation, s, may be estimated from the coverage of diffusion: $s = c \cdot kn/(mn) = ck/m$.

In order to estimate the relationship between bandwidth consumption and the degree of saturation s, we first seek to examine the relationship between bandwidth consumption and the coverage of diffusion. Consider a particular symbol being diffused, x. Let b_x be the bandwidth consumption of diffusing x, *i.e.*, the number of times that x is broadcasted in the controlled diffusion process. Let c_x be the coverage of x, *i.e.*, the number of nodes that have x at the end of the diffusion process.

Recall that we have estimated the number of new nodes being covered by a re-broadcast as 0.42ρ. It follows that $c_x = (b_x - 1)0.42\rho + \rho = 0.42b_x\rho + 0.58\rho$. If the total bandwidth consumption of diffusing all symbols of α is $b = b_x \cdot kn$, we have the following relationship between the total bandwidth consumption and the coverage of diffusion: $c = 0.42b\rho/(kn) + 0.58\rho$. Substituting the derived c in $s = ck/m$, we then have

$$s = \frac{\rho}{m}\left(\frac{0.42b}{n} + 0.58k\right).$$

The above estimate suggests that, in order to achieve a certain degree of saturation, the total bandwidth consumption of diffusion should be proportional to the number of symbols to be diffused, as well as to the normalized size of the network, *i.e.*, the area of deployment of the network divided by the disk area within the communication range of a node.

2.5 Saturation vs. Gleaning Time

We now consider the case where the controlled diffusion process of the data item α has completed, and over a certain period of time, the diffused symbols have mingled uniformly in the network, due to the random trajectories of mobile nodes. Under such an assumption, we estimate the expected gleaning time of a consumer, should it now generates a request for α. We first consider a new consumer that has just joined the network, and then modify our estimate for a consumer that was previously in the network.

To facilitate our analysis, we assume that the decoding inefficiency in Tornado coding is 1, *i.e.*, exactly n symbols is required to recover α. We further assume that the requesting consumer can obtain any useful symbols from the surrounding nodes, once they become neighbors to one another. In order to glean n unique symbols from its neighborhood, the number of symbols the requesting consumer is expected to encounter is:

$$\sum_{i=0}^{n-1} \frac{c(kn)}{c(kn) - ci} = \sum_{i=0}^{n-1} \frac{kn}{kn - i} \approx \frac{kn}{k - \frac{1}{2}} \qquad (1)$$

Therefore, the number of nodes the consumer is expected to encounter can be estimated as:

$$\frac{1}{ns}\frac{kn}{k-\frac{1}{2}} = \frac{k}{(k-\frac{1}{2})s}.$$

It follows that the expected gleaning time, t_g, is

$$t_g \approx \max\left\{(\frac{k}{(k-\frac{1}{2})s} - \rho)\Delta t, 0\right\} + t_{tr}.$$

where t_{tr} is the net transfer time of the symbols. Note that the above result is an *overestimate* in that it does not take into account the fact that symbols found at the same reservoir node are distinct; it is an *underestimate* in that the $\approx$ in Eq. (1) is actually $\geq$, and in that it ignores the time overhead it takes for the consumer to set up connections with its neighbors. However, the result should still provide insights on how the gleaning time may be related to the degree of saturation. It shows that, when saturation is beyond a certain level such that a node is able to collect sufficient symbols from one set of neighbors, then the gleaning time is dominated by the transfer time of the symbols. However, if saturation is below such a level, then the consumer needs to spend time in both receiving the symbols and in waiting to meet new neighbors. Since the second term is on the scale of physical movement, it usually dominates the gleaning time. Naturally, the existence of such a dominating factor depends on whether the number of nodes that the consumer needs to encounter is larger than the size of its local neighborhood. On the other hand, the expected number of nodes to be encountered by a consumer is inversely proportional to the degree of saturation.

If the consumer is not new and has been previously present in the network, the expected gleaning time is smaller, since the consumer may very well be a reservoir node itself, and may have accumulated a number of symbols during the diffusion process before its request arrives. We can therefore adjust the above estimate to

$$t_g \approx \max\left\{(\frac{k}{(k-\frac{1}{2})s} - \rho - 1)\Delta t, 0\right\} + t_{tr}.$$

The relationship between t_g and saturation s is similar to the case where the consumer is a new node in the network.

2.6 iFlow vs. Unicast

The total bandwidth consumption of accessing a data item α using iFlow is approximately $b + n(mq_\alpha)$, where the first term represents the bandwidth consumption of diffusion, and the second term is the bandwidth consumption of gleaning. In comparison, the total bandwidth consumption of accessing α using the all-unicast approach may be estimated as $nh(mq_\alpha)$, where h is the expected number of hops between a pair of arbitrary nodes within the network. Therefore, in order to satisfy the minimum design requirement that iFlow should be more bandwidth efficient than the plain unicast approach, we need to satisfy $b < nmq_\alpha(h-1)$. This upper bound may be denoted as b_u.

On the other hand, the delay requirement of the application defines a lower bound on the coverage of diffusion. Since coverage is controlled by bandwidth consumption, we have a corresponding lower bound for b, b_l. For any value of b such that $b_l \leq b \leq b_u$, the delay requirement is expected to be satisfied, while the bandwidth efficiency is expected to be better than the all-unicast approach. The supplier has the choice of trading delay for less bandwidth consumption, by choosing b that is closer to b_l; or, alternatively, trading bandwidth for a smaller delay, by choosing b that is closer to b_u. However, note that the average memory overhead on iFlow nodes is proportional to saturation. Therefore, the former choice is usually more preferable, since it leads to lower saturation and hence lower memory overhead.

Our analysis shows that the lower bound b_l is independent of the popularity q_α, while the upper bound b_u is proportional to q_α. Therefore for the range $[b_l, b_u]$ to be non-empty, we have a lower bound requirement on q_α. This confirms the intuition that iFlow may be more bandwidth efficient when more consumers request a popular data item. However, this requirement is rather loose in many scenarios; that is, contrary to common intuitions, the iFlow algorithms may *both* achieve bandwidth efficiency *and* meet the delay requirement for data items that are not popular, depending on the network characteristics. We proceed to show such an example.

Consider a moderately dense network with total number of nodes $m = 300$, average node degree $\rho \approx 17$, and average number of distance between a pair of nodes $h \approx 4$ hops. Assume that the data item of interest, α, has $n = 100$ symbols and Tornado codes with a stretch factor $k = 2$ is used. We now consider the extreme case where $mq_\alpha = 1$, *i.e.*, only one node in the network has a request for α. We show that if we choose the reach of diffusion to be just 1, then both the lower bound requirement and the upper bound requirement can be satisfied. When the reach is one, $b = kn = 2n < 3n = n(mq_\alpha)(h-1)$, therefore the upper bound requirement due to bandwidth efficiency is satisfied. On the other hand, the saturation $s = ck/m = (\rho+1)k/m = 0.12$; therefore the first term in our estimate of the gleaning time, $\max((\frac{k}{(k-\frac{1}{2})s} - \rho - 1)\Delta t, 0) = 0$ since $\frac{k}{(k-\frac{1}{2})s} \approx 11 < 18 = \rho + 1$. This means that the gleaning time is dominated by the symbol transfer time, and should therefore satisfy any reasonable delay requirement.

Therefore we conclude that there exist abundant opportunities for improving bandwidth efficiency using the information rendezvous algorithms proposed in iFlow, even with non-popular data items and relatively strict application requirements on delay, given that node mobility is present.

2.7 iFlow: The Multiple-Supplier Case

We now briefly consider the case where complete copies of the data item of interest, α, exist on *multiple suppliers* before the diffusion process. Such a scenario may exist if, for example, by the time that the popularity of a data item popularity is estimated to be sufficiently high to initiate diffusion, a few nodes have already acquired α through unicast.

The proposed iFlow algorithms adapt to the multiple-supplier case naturally without modifications. However, to maintain the same level of bandwidth consumption, each supplier may choose to use a small reach as its control parameter. Such a case enjoys two advantages over the single-supplier case. First, since the reach parameters used are small, in many cases just one, there exists less overlap among different broadcasts; the same amount of bandwidth consumption may now lead to a larger coverage. Second, reservoir nodes covered by diffusion has higher diversity in terms of geographical distribution; this observation, again, assists the reservoir nodes to mingle with the regular nodes more promptly and rapidly. We conclude that iFlow performs better in the multiple-supplier case, which conforms to the intuition.

3 Improving iFlow: Network Coding

The ultimate objective in the design of the iFlow architecture is to facilitate bandwidth-efficient information access. In this section, we present the important concepts of network coding [7] in wireline networks, propose to apply network coding on reservoir nodes to further improve bandwidth efficiency in iFlow.

Network coding is a theoretical strategy that has been proposed in the area of information theory [7, 8, 9], the objective of which is to increase end-to-end throughput in multicast sessions in wireline networks, which is subtly different from our goal of improving bandwidth efficiency (*i.e.,* delivering more useful data with limited channel capacity) in wireless networks. With network coding, bits of data to be delivered are not merely treated as "atoms" that may *only* be replicated and forwarded at intermediate nodes; rather, data may be *coded* before being forwarded further.

Similar to Tornado codes on information suppliers, bitwise exclusive-or ($\oplus$) can be employed as the basic coding operation for its computational efficiency. Different from Tornado codes, network coding may be used not only at the source node, but also at intermediate nodes; it may code not only information of the same data item, but independent information from different data items as well. Coded data may be decoded by a downstream or destination node, based on its knowledge of the coding strategy.

3.1 Network Coding: a Review of Concepts

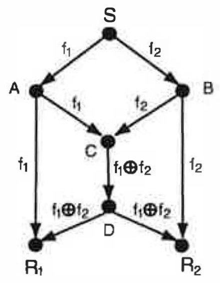


Figure 4: The effectiveness of network coding in wireline networks

We briefly review the concepts of network coding with an example shown in Fig. 4 [7]. The example shows how the session throughput of an 1-to-2 multicast session may be improved in wireline networks. In the figure, f_1 and f_2 represent two independent information flows originating from the source S. Node C transmits the coded flow $f_1 \oplus f_2$ along the "bottleneck" link CD to node D, which then forwards the coded flow to both destinations R_1 and R_2. R_1 and R_2 can recover $\{f_1, f_2\}$ from $\{f_1, f_1 \oplus f_2\}$ and $\{f_2, f_1 \oplus f_2\}$, respectively. The session achieves a throughput of $2C$, assuming each link has capacity C. Without network coding, it can be verified that the achievable throughput is only $3C/2$.

3.2 Network Coding in Wireless Networks

While network coding may increase throughput of multicast sessions in wireline networks, there exist fundamental differences between wireless and wireline networks. With respect to bottleneck formation, wireline networks are *link-centric*, while wireless networks are *node-centric*. In wireline networks, a single link can form a bottleneck due to limited link capacity, while a forwarding node is usually not a concern; in wireless networks, a single node is sufficient to form a bottleneck, since virtual links sharing the same node may not transmit concurrently, [10]. We are not aware of any previous work that has studied the problem of applying network coding in wireless networks, especially when the objective is to improve bandwidth efficiency.

We believe that network coding may still be applied effectively in wireless networks, in order to improve bandwidth efficiency. Fig. 5 shows an example in which network coding helps to reduce the total bandwidth consumption in two 1-to-2 wireless multicast sessions, where S_1 and S_2 are the sources and R_1 and R_2 are the destinations. Without network coding, four transmissions are required to deliver a packet from each source

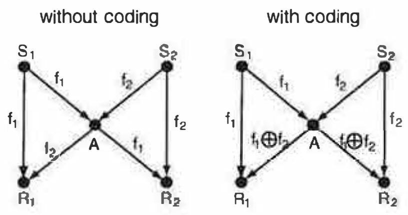


Figure 5: Network coding in wireless networks: an example

to each destination: $S_1 \xrightarrow{f_1} (A, R_1)$, $S_2 \xrightarrow{f_2} (A, R_2)$, $A \xrightarrow{f_2} R_1$ and $A \xrightarrow{f_1} R_2$. With network coding, the "bottleneck" node A transfers the coded flow $f_1 \oplus f_2$ to both receivers at once, only three transmissions are necessary: $S_1 \xrightarrow{f_1} (A, R_1)$, $S_2 \xrightarrow{f_2} (A, R_2)$, and $A \xrightarrow{f_1 \oplus f_2} (R_1, R_2)$. Therefore overall bandwidth efficiency is improved by 1/3.

3.3 Network Coding: Improving iFlow

We proceed to discuss the details of improving the bandwidth efficiency in the iFlow architecture by applying network coding on reservoir nodes. A major feature of iFlow as opposed to a generic wireless ad hoc network is that, iFlow applies Tornado codes in information suppliers before the diffusion process. We observe that, both network coding and Tornado codes employ the bitwise exclusive-or operation, and therefore can act in concert with each other naturally within the iFlow framework, as we will show shortly. In iFlow, a reservoir node has the opportunity to apply network coding when it is relaying multiple symbols during diffusion, supplying symbols for multiple consumers during gleaning, or a mixture of the two. In these cases, the reservoir node forms a "bottleneck" node that may apply take advantage of the broadcast nature of wireless transmission by broadcasting a recoded symbol that may potentially benefit more neighbors. Due to limit of space, we show an example of applying network coding in information diffusion only.

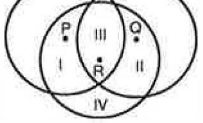


Figure 6: Network coding in information diffusion: an example

Assume that during diffusion, a node R receives a symbol a from node P and a symbol b from node Q, respectively, as shown in Fig. 6. If R plans to further relay a symbol in the diffusion process, it may perform one of the following tasks: (1) re-broadcast a or b directly; or (2) re-broadcast the recoded symbol $a \oplus b$, in accordance with the concept of network coding. We argue that, re-broadcasting $a \oplus b$ is a better choice, since it can potentially benefit more neighboring nodes of R.

Table 4: Network coding in information diffusion: a comparison

	current symbols	broadcast a at R	broadcast b at R	broadcast $a \oplus b$ at R
I	a	a	a, b	a, b
II	b	a, b	b	a, b
III	a, b	a, b	a, b	a, b
IV	ϕ	a	b	$a \oplus b$

Table 4 shows the set of symbols acquired by nodes within different areas, before and after R's re-broadcasting of a, b, or $a \oplus b$. The choice at R does not affect nodes within area III, since they have already received both a and b before R's re-broadcasting. If R re-broadcasts a, then nodes within area II and IV will benefit, since a is new to them; nodes within area I will not benefit since they have received a already. Similar arguments apply if R re-broadcasts b. In comparison, if R re-broadcasts the recoded symbol $a \oplus b$ based on network coding, then nodes within both area I and area II will benefit and obtain complete information on both symbols a and b.

It is not obvious, though, whether nodes in area IV also benefit from the coded transmission from R. Naturally, if nodes within area IV received a or b previously from other nodes, or will receive a or b later on, the value of $a \oplus b$ can then be realized. Furthermore, since a and b are being diffused concurrently within nearby network areas, it is highly probable that they are coded symbols of the same data item α. In that case, the recoded symbol $a \oplus b$ has its own value without being recovered to a and b first. The reason is that, same as a or b, $a \oplus b$ is just another coded symbol obtained by applying the $\oplus$ operations over certain data segments in α, and therefore can be transmitted within the network and be used as input to the Tornado decoding procedure as well.

To further illustrate such harmony between Tornado codes and network coding, consider the following example. For clarity, assume that there are only three data segments in α, 1, 2 and 3. Further, assume that a Tornado coding scheme with a stretch factor of 2 is used, with 1, 2, 3, $1 \oplus 2$, $2 \oplus 3$ and $1 \oplus 2 \oplus 3$ being the coded symbols. We can verify that any combination of three distinct symbols is sufficient to recover the data item with probability 0.8, and any combination of four distinct symbols is sufficient for the recovery with probability 1. Therefore such a coding scheme has the decoding inefficiency of $(3 \times 0.8 + 4 \times 0.2)/3 = 1.07$. If symbol a in the previous example is, say, $1 \oplus 2$, and symbol b is $2 \oplus 3$, then the recoded symbol resulting from network coding, $a \oplus b$, is precisely $1 \oplus 3$, which intuitively also contains useful information for the purpose of recovering a. We then come to the conclusion that re-broadcasting $a \oplus b$ at R is more bandwidth efficient than re-broadcasting a or b with a high probability. This example shows the advantage of network coding in the diffusion process.

To conclude, though iFlow is a complete architecture and set of algorithms to enable bandwidth-efficient information access, further improvements on bandwidth efficiency can be realized if network coding is applied appropriately in the information diffusion process.

4 iFlow: Simulation

For the purpose of evaluating the performance of various aspects of the iFlow architecture and the feasibility of its deployment, we performed simulations of the iFlow algorithms in C++, followed by a prototype implementation of iFlow as a COM-based middleware layer (discussions of which are postponed to Sec. 5).

4.1 Bandwidth Efficiency

The primary design objective of the iFlow architecture is to enable bandwidth-efficient information access within mobile ad hoc applications, given a certain degree of user mobility and information popularity. Our analysis in Sec. 2 has shown that, there exist abundant opportunities that iFlow can achieve this goal. This observation is verified by forthcoming empirical results.

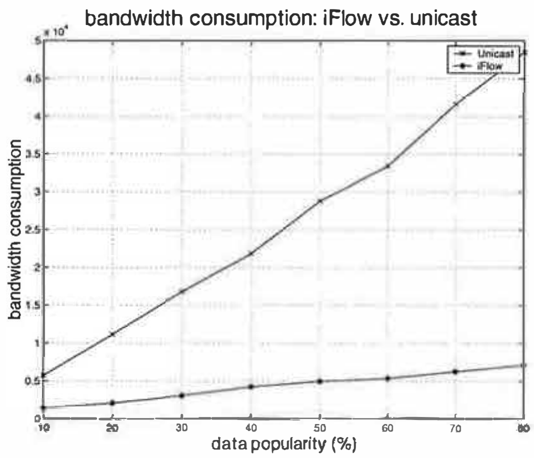


Figure 7: Bandwidth efficiency under different degrees of data popularity: iFlow vs. unicast

Fig. 7 shows a comparative study between the iFlow algorithms and the plain all-unicast approach, with respect to bandwidth consumption of disseminating the same data item. In the comparison, we simulate with the same network environment and the same simulation framework. The simulation parameters are established as follows. (1) The area of deploying the mobile ad hoc network is 500m $\times$ 500m. (2) The total number of users m is 300. (3) The communication range of each node R is 70m. (4) The number of uncoded data segments in the data item $n = 50$. (5) With respect to Tornado codes, the stretch factor of Tornado codes k is 3, and the decoding inefficiency is 1.0. (6) The nodes move around the deployment area using the random way-point mobility model, with the pause time as 0 seconds, and the velocity as 9 m/s. (7) The expected waiting time $E[T_x] = t_s$ between consecutive broadcasts at supplier is 2 seconds; while the expected waiting time $E[T_c]$ between consecutive probes at consumers is 3 seconds.

With respect to the bandwidth consumption in iFlow, we take the bandwidth consumed in both diffusion and gleaning processes into account. For the all-unicast approach, we compute the total length of routes between the supplier and the consumers. In cases that the supplier and the consumer are separated within different partitions of the network, we wait until they move into the same partition.

Simulation results in Fig. 7 show that, the total bandwidth consumption of the all-unicast approach grows linearly with data popularity, and is always larger than the bandwidth consumption of the iFlow system. The difference becomes more significant as popularity grows, up to an order of magnitude. The reason behind this observation is that, for the all-unicast approach, bandwidth consumption is proportional to data popularity; while for iFlow, the total bandwidth consumption is a summation over two terms: diffusion bandwidth consumption and gleaning bandwidth consumption. The first term remains at a constant level regardless of data popularity, only the second term grows linearly with popularity. Since information gleaning consists of one-hop transmissions only, the second term is much smaller than the total bandwidth consumption of multi-hop unicast transmissions. Therefore, the bandwidth consumption of iFlow grows much slower than that of unicast.

4.2 Diffusion Bandwidth Consumption vs Saturation

In Sec. 2, we have analyzed the relationship between bandwidth consumption and expected gleaning time by first deriving the relationship between diffusion bandwidth consumption and saturation, and then deriving the relationship between saturation and gleaning time. Recall that the theoretical result of our analysis on the first relationship is $s = \rho \cdot (0.42b/n + 0.58k)/m$. Below we compare results from our simulations to such a theoretical estimate.

We have performed two sets of simulations, one of which has a deployment size of 500m $\times$ 500m, while

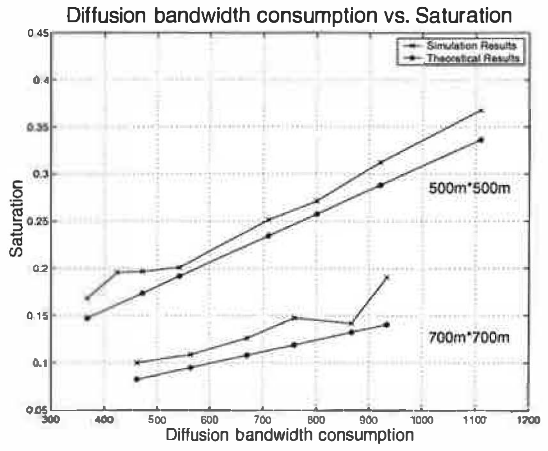


Figure 8: The relationship of diffusion bandwidth consumption and saturation

the other has a deployment size of 700m × 700m. We vary the reach r from 2 to 4, and vary the relay probability p from 0.1 to 0.2 and from 0.3 to 0.4 for the 500m × 500m case and the 700m × 700m case, respectively. Other parameters remain unchanged as in Sec. 4.1. We choose different ranges of relay probabilities for the two cases so that saturation within two networks under the same bandwidth consumption can be compared.

Both simulation and previous analytical results are shown in Fig. 8. It shows that overall, results from our simulations agree with our theoretical analysis. In particular, for the same amount of bandwidth consumption, the degree of saturation in the 500m × 500m network is approximately twice as high as that in the 700m × 700m network. This confirms our previous observation that, in order to achieve the same level of saturation, bandwidth consumed in diffusion should be proportional to the normalized network size.

4.3 The Role of Relay Probability

The relay probability p plays an important role in the iFlow algorithms. It is used to arbitrate the trade-off between bandwidth efficiency and the delay of satisfying requests from the application. With the similar network setup[2] carried forward from the previous experiments, Fig. 9 has shown how bandwidth consumption and gleaning time vary as the relay probability varies. As we may observe, as p grows, bandwidth consumption increases and gleaning time decreases. However, when the value of p reaches a certain level (0.5 in this case), further increases of p elevates the amount of bandwidth consumption without significant effects on the gleaning time. This suggests that the combination of a smaller relay probability with a larger reach is a better choice, rather than the combination of a larger relay probability with a smaller reach. This confirms the corresponding statements in our analysis of the algorithms (Sec. 2). As previously explained, this is due to the fact that the latter choice introduces more overlap among the broadcasts during the diffusion, and is therefore less cost-effective.

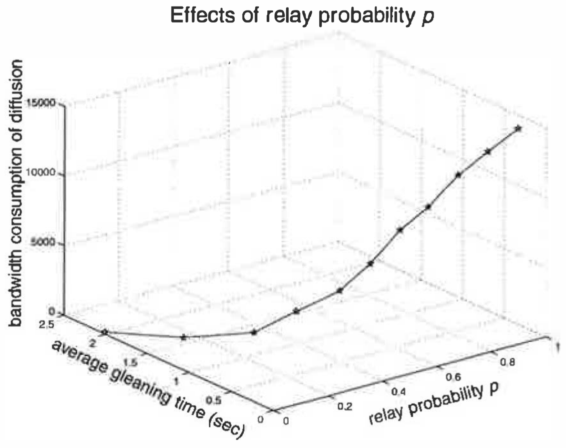


Figure 9: Effects of varying the relay probability p in the controlled diffusion process

4.4 Comparison of Delay Latency

Finally, we perform simulations to compare the delay latency of performing the same data dissemination task using unicast, iFlow, and iFlow with Tornado coding replaced by cyclic repetition. The results are presented in Table 5, using the same network simulation parameters as the first experiment[3]. First, we observe that, by introducing Tornado coding on information suppliers, the latency of satisfying requests experienced by the mobile ad hoc application is dramatically reduced. The justification is that, as we have explained, erasure codes such as Tornado coding gracefully solves the problem of obtaining the last few segments of data, from which the scheme of cyclic repetition suffers. Second, we notice that although delay of unicast is smaller than that of iFlow, they are on the same magnitude. This is due to the poor data availability of the unicast approach, since a consumer node may be partitioned from the only supplier node when its request arrives. In that case, the consumer has to wait until it moves into the same network partition as the supplier. In denser networks where partition rarely occurs, the performance of unicast should be better, in terms of delay latency.

The set of simulation results presented in this section has verified the insights we have obtained from the analysis of iFlow, and has provided solid proof that iFlow

[2]It is identical except that the reach of diffusion is 5, the user velocity is 2 m/s, and $E(T_x) = 1$ second.

[3]With the exception that the deployment size of the network is 600m × 600m, the transmission range $R = 50$ m, the reach is 2, and the relay probability $p = 0.1$.

Table 5: Latency: Tornado coding vs. cyclic repetition

user velocity (m/s)	3	5	7	9
Delay: Unicast(s)	2.3	2.0	1.4	1.5
Delay: iFlow (s)	4.3	4.0	3.4	3.6
Delay: cyclic repetition (s)	11.8	8.8	7.4	6.0

is able to consistently outperform the alternatives without iFlow, with respect to both bandwidth efficiency and latency.

5 iFlow: Implementation

Beyond simulation results previously shown, we have implemented the iFlow architecture as a middleware layer based on the Microsoft Component Object Model (COM), supporting COM-aware applications on the Windows platform. The iFlow middleware layer exposes COM interfaces for the applications to invoke, and delivers COM events to applications, so that the application may implement customized event handlers to handle iFlow events. For basic communication between neighboring nodes, the iFlow middleware utilizes the Windows Sockets library available on Windows. In order to realize a wireless ad hoc network, we have further used the *ad hoc mode* of IEEE 802.11b wireless LAN as the MAC and physical substrate in our testbed, without fixed access points. We employ Windows-based laptop computers for our testbed, mainly with Windows 2000 and Windows XP as operating systems.

The advantage of using COM as our middleware substrate is to support the ubiquitous availability of Windows applications, including those on Pocket PC based platforms. The entire set of iFlow algorithms are built as a Dynamic Link Library (DLL) in Windows, to be readily loaded by any COM-aware applications. The availability of Rapid Application Development tools such as Microsoft Visual Basic greatly facilitates making the necessary modifications to existing applications to take advantage of iFlow, if there are interests for iFlow events to be handled. Using Visual Basic, we have implemented a prototype application to employ the services of the iFlow middleware, with graphical user interfaces for the purpose of illustrating the status of iFlow in action.

For the purpose of showing global properties of the entire wireless ad hoc network, we have resorted to the creation of an *omniscient observer*, which, obviously, may not be available in real-world scenarios. However, the availability of the omniscient observer in our implementations greatly facilitates the monitoring of instantaneous network and node states, such as total number of nodes, node locations, roles of different nodes (consumers, reservoir nodes or information suppliers), as well as the number of consumers that have already reconstructed the requested data item. Due to the unavailability of GPS devices and the fact that most of our tests are conducted indoors, we have simulated the node locations on the omniscient observer, and then delivered the node locations to participating nodes on a periodic basis. The other advantage of such simulated locations is that the degree of node mobility may be easily varied, leading to more deterministic studies of the effects of mobility[4].

The implementation of iFlow middleware layer is designed to be multi-threaded in order to accommodate multiple incoming requests and ongoing connections. There exists three types of threads: (1) the main thread to handle COM-based method invocations; (2) the server thread to listen on the well-known port and create TCP-based stream sockets; and (3) the "worker" threads to process incoming requests. We have accomplished challenging tasks of COM-based multi-threaded programming, where the COM interface pointer needs to be marshaled per thread.

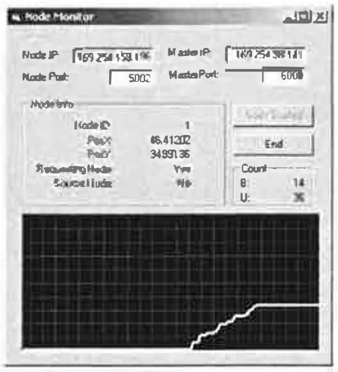
With respect to the exposed interfaces for the application to invoke, and the delivered COM events for the application to handle, we briefly show precise definitions of example methods and events in the iFlow interface, defined in the Microsoft Interface Definition Language (IDL):

```
interface iFlowWrapper : IDispatch
{
  typedef struct tagPacket { ... } Packet;

  \\ methods
  \\ send a packet for local broadcast through iFlow
  HRESULT BroadcastPacket([in] Packet* Msg,
                   [out,retval] int* pCount);
  \\ initialize the iFlow middleware algorithms
  HRESULT StartServer([in] BSTR srcAddress,[in] int srcPort
                   [in] int IsMaster);
  \\ parse incoming message
  HRESULT ParseMessage([in] int pPacket,[in] UINT pSocket,
                   [out, retval] int* pVal);
  ...
};

\\ events
dispinterface _iFlowWrapperEvents
{
  HRESULT OnDiffuse();
  HRESULT OnRecvBroadcast([in] short PacketNumber);
  HRESULT OnReBroadcast([in] short PacketNumber,
                        [in] short reach);
  ...
};
```

To demonstrate the results of such an implementation, we have deployed iFlow in a 30-node network with one data item of interest. Fig. 10 shows graphical user interfaces on a regular node and the omniscient observer

[4]Since node mobility is simulated, it may be more appropriate to refer to our testbed as an *emulation* testbed rather than implementation. However, with modest modifications, we believe that the implementation of iFlow may still be readily be deployed in real-world scenarios.

during a diffusion session. The results have mostly agreed with our simulations, which we choose not to show repeatedly. For completeness, Fig. 11(a) shows the per-node progress recorded within iFlow on each of the nodes (10 nodes are shown as examples), and Fig. 11(b) shows the number of consumers that have reconstructed the requested data item over time. To conclude, even though the implementation is a proof-of-concept prototype, it has demonstrated the feasibility of real-world deployment of iFlow on wireless devices; we have especially shown ready support for COM-based Windows applications that are ubiquitously available.

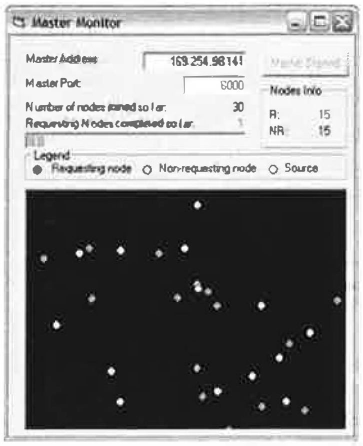


Node IP: 169.254.158.196 Master IP: 169.254.98.141
Node Port: 5002 Master Port: 6000

Node Info
Node ID: 1
PosX: 65.41202
PosY: 34.99135
Requesting Node: Yes
Source Node: No

B: 14
U: 36

(a) The monitoring graphical user interface of the *iFlow* middleware architecture on regular nodes, showing the number of symbols accumulated so far. (Windows 2000)



Master Address: 169.254.98.141
Master Port: 6000
Number of nodes joined so far: 30
Requesting Nodes completed so far: 1

Nodes Info
R: 15
NR: 15

(b) The monitoring graphical user interface of the *iFlow* architecture on the "omniscient observer", showing a global view of the entire network. (Windows XP)

Figure 10: The middleware implementation of iFlow architecture: controlled diffusion session in action

6 Related Work

The design of the iFlow architecture and algorithms has been inspired by various exciting work from recent literature. We position iFlow in light of these work and highlight our original contributions in comparative studies.

Byers *et al.* studied the problem of delivering bulk data (on the order of gigabytes) to a large number of users across the Internet through an overlay network [11]. Their approach is similar to ours in that both employ Tornado codes to resolve the problem of obtaining the final data segments, as well as to provide robustness. Also, intermediate nodes have recoding capability and collaborate with each other actively. However, the focus of their work is on the problem of reconciliation between peer nodes, which is complicated by the application of Tornado codes. The design objective of their system is to deliver content to end users in a timely fashion in broadband wireline networks, while the design of iFlow focuses on improving bandwidth efficiency of information access in mobile ad hoc networks.

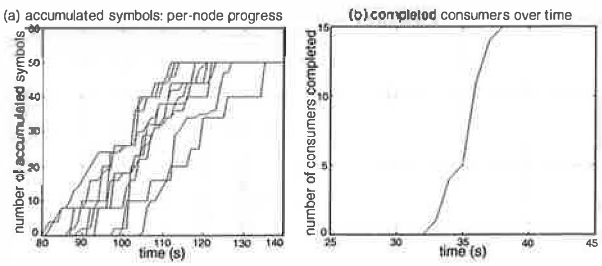


Figure 11: Live progress on consumers with the iFlow middleware testbed

The theoretical work of Grossglauser *et al.* [12] first reveals the fact that *node mobility*, which is usually treated as a negative factor in wireless networking, can play a *positive* role. Their main result is that, when nodes are mobile, the available end-to-end capacity for each source-destination pair in the network can remain constant, rather than approaches zero, as the size of the network grows. Although this result remains largely theoretical due to its strong assumptions, it does suggest that in certain scenarios in practice, it is possible to devise algorithms to trade off delay for better network performance, which is throughput in their case, and bandwidth efficiency in ours.

Network coding was first proposed and studied by Ahlswede *et al.* in the context of wireline networks [7]. It is shown that, applying network coding (usually linear codes suffice[8]) on intermediate nodes over a multicast network may increase its capacity. The problem of whether a given throughput can be achieved in a given multicast network was studied subsequently using an algebraic approach, with sufficient and necessary conditions provided for some cases [9]. Although exciting insights are provided, the existing studies on network coding has remained to be largely theoretical, and we are not aware of any published work that studies network coding in wireless networks. Due to the unique spatial contention of wireless transmissions in the local neighborhood, the effects of network coding is dramatically different from its counterpart in broadband wireline networks. In Sec. 3, we have shown that network coding can lead to higher bandwidth efficiency in wireless transmissions as well.

The 7DS system proposed by Papadopouli *et al.* [4] presents a practical study of data sharing among nodes in an ad hoc network that is sparse and less mobile. A node that loses Internet connection may acquire a desired data item from its neighbors, if it has been cached by one or more of them. The authors focus on the issues of how various network dynamics and design choices affect the performance of the 7DS protocol. The design of 7DS concentrates on data availability rather than bandwidth efficiency. With iFlow, although data availability is also improved due to its nature of distributed content caching, our main interests are on utilizing node mobility to disseminate popular data items in a bandwidth efficient way, subjecting to delay requirements imposed by applications.

A recent short paper by Goel *et al.* has first proposed to use Tornado codes to facilitate data dissemination in wireless networks [13]. Their simulation results show that the time it takes for all requesting nodes to obtain the data item using Tornado codes may be significantly reduced compared to not using Tornado codes. However, there does not exist any analytical work to support the results, and the results are limited to pre-defined mobility models. Further, the paper did not examine the issue of bandwidth efficiency in wireless networks. In the design of iFlow, we have brought the separate pieces together, including the use of Tornado codes that was previously mentioned [13], and also the algorithms facilitating information rendezvous and network coding on third-party nodes. We design and assemble the strategies with one unified objective: improving bandwidth efficiency, and study the effects of various tradeoffs and parameters pertaining this goal. We have not been able to identify such analysis in previous studies.

7 Concluding Remarks

This paper has presented the architecture, algorithms and analysis of *iFlow*, a middleware framework to facilitate information access in mobile ad hoc applications. We have shown that, iFlow is able to transparently provide a bandwidth-efficient way of information flow from suppliers to consumers, with strategies that include information rendezvous, erasure codes and network coding. We note that a high degree of node mobility actually contributes to achieving and improving bandwidth efficiency, and a relaxed delay requirement in delay-insensitive applications is an ideal scenario to deploy iFlow.

We are convinced that the full potential of iFlow with respect to efficient information access has yet to be realized. As an example, we may devise a mechanism for requests from consumers to be self-routed to the reservoir nodes (or suppliers) that hold the missing symbols, so that the requests may be satisfied earlier, with a slight penalty on bandwidth efficiency. Such a mechanism may be invoked when the consumers are about to activate the panic mode to directly contact the suppliers. Other improvements are also possible, including more in-depth integration of Tornado codes and network coding. We believe that, by extending our prototype implementation of iFlow as a middleware layer, iFlow may be rapidly deployed to assist emerging applications in mobile ad hoc networks, and, subsequently, redefine the communication patterns of such applications. Such patterns may further be studied to facilitate the design of lower-layer ad hoc network protocols.

References

[1] J. Li, C. Blake, D. Couto, H. Lee, and R. Morris, "Capacity of Ad Hoc Wireless Networks," in *Proc. of ACM MobiCom*, September 2001, pp. 61–69.

[2] D. Estrin, L. Girod, G. Pottie, and M. Srivastava, "Instrumenting the World With Wireless Sensor Networks," in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, 2001.

[3] J. Chang and L. Tassiulas, "Energy Conserving Routing in Wireless Ad hoc Networks," in *Proceedings of IEEE INFOCOM*, 2000.

[4] M. Papadopouli and H. Schulzrinne, "Effects of Power Conservation, Wireless Coverage and Cooperation on Data Dissemination among Mobile Devices," in *Proc. of ACM MobiHoc*, 2001.

[5] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A Digital Fountain Approach to Reliable Distribution of Bulk Data," in *Proc. of ACM SIGCOMM*, 1998, pp. 56–67.

[6] J. Wieselthier, G. Nguyen, and A. Ephremides, "On the Construction of Energy-Efficient Broadcast and Multicast Trees in Wireless Networks," in *Proc. of IEEE INFOCOM*, 2000.

[7] R. Ahlswede, N. Cai, S. R. Li, and R. W. Yeung, "Network Information Flow," *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1204–1216, July 2000.

[8] S.-Y. R. Li and R. W. Yeung, "Linear Network Coding," *IEEE Transactions on Information Theory, to appear*, 2002.

[9] R. Koetter and M. Medard, "Beyond Routing: An Algebraic Approach to Network Coding," in *Proc. of IEEE INFOCOM*, 2002.

[10] V. Bharghavan, A. Demers, S. Shenker, and L. Zhang, "MACAW: A Media Access Protocol for Wireless LANs," in *Proc. of ACM SIGCOMM*, 1994, pp. 212–225.

[11] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed Content Delivery Across Adaptive Overlay Networks," in *Proc. of ACM SIGCOMM*, 2002.

[12] M. Grossglauser and D. Tse, "Mobility Increases the Capacity of Ad-hoc Wireless Networks," in *Proc. of IEEE INFOCOM*, 2001.

[13] S. K. Goel, M. Chai, D. Xu, and B. Li, "Efficient Peer-to-Peer Data Dissemination in Mobile Ad-hoc Networks," in *Proc. of International Workshop on Ad Hoc Networking*, August 2002.

Full TCP/IP for 8-Bit Architectures

Adam Dunkels
Swedish Institute of Computer Science
adam@sics.se, http://www.sics.se/~adam/

Abstract

We describe two small and portable TCP/IP implementations fulfilling the subset of RFC1122 requirements needed for full host-to-host interoperability. Our TCP/IP implementations do not sacrifice any of TCP's mechanisms such as urgent data or congestion control. They support IP fragment reassembly and the number of multiple simultaneous connections is limited only by the available RAM. Despite being small and simple, our implementations do not require their peers to have complex, full-size stacks, but can communicate with peers running a similarly light-weight stack. The code size is on the order of 10 kilobytes and RAM usage can be configured to be as low as a few hundred bytes.

1 Introduction

With the success of the Internet, the TCP/IP protocol suite has become a global standard for communication. TCP/IP is the underlying protocol used for web page transfers, e-mail transmissions, file transfers, and peer-to-peer networking over the Internet. For embedded systems, being able to run native TCP/IP makes it possible to connect the system directly to an intranet or even the global Internet. Embedded devices with full TCP/IP support will be first-class network citizens, thus being able to fully communicate with other hosts in the network.

Traditional TCP/IP implementations have required far too much resources both in terms of code size and memory usage to be useful in small 8 or 16-bit systems. Code size of a few hundred kilobytes and RAM requirements of several hundreds of kilobytes have made it impossible to fit the full TCP/IP stack into systems with a few tens of kilobytes of RAM and room for less than 100 kilobytes of code.

TCP [21] is both the most complex and the most widely used of the transport protocols in the TCP/IP stack. TCP provides reliable full-duplex byte stream transmission on top of the best-effort IP [20] layer. Because IP may reorder or drop packets between the sender and the receiver, TCP has to implement sequence numbering and retransmissions in order to achieve reliable, ordered data transfer.

We have implemented two small generic and portable TCP/IP implementations, *lwIP* (lightweight IP) and *uIP* (micro IP), with slightly different design goals. The lwIP implementation is a full-scale but simplified TCP/IP implementation that includes implementations of IP, ICMP, UDP and TCP and is modular enough to be easily extended with additional protocols. lwIP has support for multiple local network interfaces and has flexible configuration options which makes it suitable for a wide variety of devices.

The uIP implementation is designed to have only the absolute minimal set of features needed for a full TCP/IP stack. It can only handle a single network interface and does not implement UDP, but focuses on the IP, ICMP and TCP protocols.

Both implementations are fully written in the C programming language. We have made the source code available for both lwIP [7] and uIP [8]. Our implementations have been ported to numerous 8- and 16-bit platforms such as the AVR, H8S/300, 8051, Z80, ARM, M16c, and the x86 CPUs. Devices running our implementations have been used in numerous places throughout the Internet.

We have studied how the code size and RAM usage of a TCP/IP implementation affect the features of the TCP/IP implementation and the performance of the communication. We have limited our work to studying the implementation of TCP and IP protocols and the interaction between the TCP/IP stack and the application programs. Aspects such as address configuration, security, and energy consumption are out of the scope of this work.

The main contribution of our work is that we have shown that is it possible to implement a full TCP/IP stack that is small enough in terms of code size and memory usage to be useful even in limited 8-bit systems.

Recently, other small implementations of the TCP/IP stack have made it possible to run TCP/IP in small 8-bit systems. Those implementations are often heavily specialized for a particular application, usually an embedded web server, and are not suited for handling generic TCP/IP protocols. Future embedded networking applications such as peer-to-peer networking require that the embedded devices are able to act as first-class network citizens and run a TCP/IP implementation that is not tailored for any specific application.

Furthermore, existing TCP/IP implementations for small systems assume that the embedded device always will communicate with a full-scale TCP/IP implementation running on a workstation-class machine. Under this assumption, it is possible to remove certain TCP/IP mechanisms that are very rarely used in such situations. Many of those mechanisms are essential, however, if the embedded device is to communicate with another equally limited device, e.g., when running distributed peer-to-peer services and protocols.

This paper is organized as follows. After a short introduction to TCP/IP in Section 2, related work is presented in Section 3. Section 4 discusses RFC standards compliance. How memory and buffer management is done in our implementations is presented in Section 5 and the application program interface is discussed in Section 6. Details of the protocol implementations is given in Section 7 and Section 8 comments on the performance and maximum throughput of our implementations, presents throughput measurements from experiments and reports on the code size of our implementations. Section 9 gives ideas for future work. Finally, the paper is summarized and concluded in Section 10.

2 TCP/IP overview

From a high level viewpoint, the TCP/IP stack can be seen as a black box that takes incoming packets, and demultiplexes them between the currently active connections. Before the data is delivered to the application, TCP sorts the packets so that they appear in the order they were sent. The TCP/IP stack will also send acknowledgments for the received packets.

Figure 1 shows how packets come from the network de-

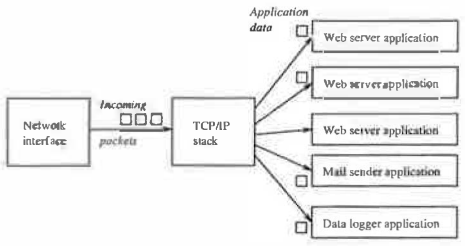


Figure 1: TCP/IP input processing.

vice, pass through the TCP/IP stack, and are delivered to the actual applications. In this example there are five active connections, three that are handled by a web server application, one that is handled by the e-mail sender application, and one that is handled by a data logger application.

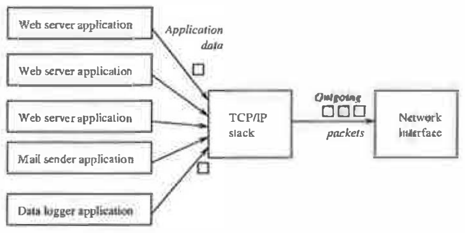


Figure 2: TCP/IP output processing.

A high level view of the output processing can be seen in Figure 2. The TCP/IP stack collects the data sent by the applications before it is actually sent onto the network. TCP has mechanisms for limiting the amount of data that is sent over the network, and each connection has a queue on which the data is held while waiting to be transmitted. The data is not removed from the queue until the receiver has acknowledged the reception of the data. If no acknowledgment is received within a specific time, the data is retransmitted.

Data arrives asynchronously from both the network and the application, and the TCP/IP stack maintains queues in which packets are kept waiting for service. Because packets might be dropped or reordered by the network, incoming packets may arrive out of order. Such packets have to be queued by the TCP/IP stack until a packet that fills the gap arrives. Furthermore, because TCP limits the rate at which data that can be transmitted over each TCP connection, application data might not be immediately sent out onto the network.

The full TCP/IP suite consists of numerous protocols, ranging from low level protocols such as ARP which translates IP addresses to MAC addresses, to application level protocols such as SMTP that is used to transfer e-mail. We have concentrated our work on the TCP and IP protocols and will refer to upper layer protocols as "the application". Lower layer protocols are often implemented in hardware or firmware and will be referred to as "the network device" that are controlled by the network device driver.

TCP provides a reliable byte stream to the upper layer protocols. It breaks the byte stream into appropriately sized segments and each segment is sent in its own IP packet. The IP packets are sent out on the network by the network device driver. If the destination is not on the physically connected network, the IP packet is forwarded onto another network by a router that is situated between the two networks. If the maximum packet size of the other network is smaller than the size of the IP packet, the packet is fragmented into smaller packets by the router. If possible, the size of the TCP segments are chosen so that fragmentation is minimized. The final recipient of the packet will have to reassemble any fragmented IP packets before they can be passed to higher layers.

3 Related work

There are numerous small TCP/IP implementations for embedded systems. The target architectures range from small 8-bit microcontrollers to 32-bit RISC architectures. Code size varies from a few kilobytes to hundreds of kilobytes. RAM requirements can be as low as 10 bytes up to several megabytes.

Existing TCP/IP implementations can roughly be divided into two categories; those that are adaptations of the Berkeley BSD TCP/IP implementation [18], and those that are written independently from the BSD code. The BSD implementation was originally written for workstation-class machines and was not designed for the limitations of small embedded systems. Because of that, implementations that are derived from the BSD code base are usually suited for larger architectures than our target. An example of a BSD-derived implementation is the InterNiche NicheStack [11], which needs around 50 kilobytes of code space on a 32-bit ARM system.

Many of the independent TCP/IP implementations for embedded processors use a simplified model of the TCP/IP stack which makes several assumptions about the communication environment. The most common assumption is that the embedded system always will communicate with a system such as a PC that runs a full scale, standards compliant TCP/IP implementation. By relying on the standards compliance of the remote host, even an extremely simplified, uncompliant, TCP/IP implementation will be able to communicate. The communication may very well fail, however, once the system is to communicate with another simplified TCP/IP implementation such as another embedded system of the same kind. We will briefly cover a number of such simplifications that are used by existing implementations.

One usual simplification is to tailor the TCP/IP stack for a specific application such as a web server. By doing this, only the parts of the TCP/IP protocols that are required by the application need to be implemented. For instance, a web server application does not need support for urgent data and does not need to actively open TCP connections to other hosts. By removing those mechanisms from the implementation, the complexity is reduced.

The smallest TCP/IP implementations in terms of RAM and code space requirements are heavily specialized for serving web pages and use an approach where the web server does not hold any connection state at all. For example, the iPic match-head sized server [26] and Jeremy Bentham's PICmicro stack [1] require only a few tens of bytes of RAM to serve simple web pages. In such an implementation, retransmissions cannot be made by the TCP module in the embedded system because nothing is known about the active connections. In order to achieve reliable transfers, the system has to rely on the remote host to perform retransmissions. It is possible to run a very simple web server with such an implementation, but there are serious limitations such as not being able to serve web pages that are larger than the size of a single TCP segment, which typically is about one kilobyte.

Other TCP/IP implementations such as the Atmel TCP/IP stack [5] save code space by leaving out certain vital TCP mechanisms. In particular, they often leave out TCP's congestion control mechanisms, which are used to reduce the sending rate when the network is overloaded. While an implementation with no congestion control might work well when connected to a single Ethernet segment, problems can arise when communication spans several networks. In such cases, the intermediate nodes such as switches and routers may be overloaded. Because congestion primarily is caused by the amount of packets in the network, and not the size of these packets, even small 8-bit systems are able to produce enough traffic to cause congestion. A TCP/IP implementation lacking congestion control mechanisms should not be used over the global Internet as it might contribute to congestion collapse [9].

Texas Instrument's MSP430 TCP/IP stack [6] and the TinyTCP code [4] use another common simplification in that they can handle only one TCP connection at a time. While this is a sensible simplification for many applications, it seriously limits the usefulness of the TCP/IP implementation. For example, it is not possible to communicate with two simultaneous peers with such an implementation. The CMX Micronet stack [27] uses a similar simplification in that it sets a hard limit of 16 on the maximum number of connections.

Yet another simplification that is used by LiveDevices Embedinet implementation [12] and others is to disregard the maximum segment size that a receiver is prepared to handle. Instead, the implementation will send segments that fit into an Ethernet frame of 1500 bytes. This works in a lot of cases due to the fact that many hosts are able to receive packets that are 1500 bytes or larger. Communication will fail, however, if the receiver is a system with limited memory resources that is not able to handle packets of that size.

Finally, the most common simplification is to leave out support for reassembling fragmented IP packets. Even though fragmented IP packets are quite infrequent [25], there are situations in which they may occur. If packets travel over a path which fragments the packets, communication is impossible if the TCP/IP implementation is unable to correctly reassemble them. TCP/IP implementations that are able to correctly reassemble fragmented IP packets, such as the Kadak KwikNET stack [22], are usually too large in terms of code size and RAM requirements to be practical for 8-bit systems.

4 RFC-compliance

The formal requirements for the protocols in the TCP/IP stack is specified in a number of RFC documents published by the Internet Engineering Task Force, IETF. Each of the protocols in the stack is defined in one more RFC documents and RFC1122 [2] collects all requirements and updates the previous RFCs.

The RFC1122 requirements can be divided into two categories; those that deal with the host to host communication and those that deal with communication between the application and the networking stack. An example of the first kind is *"A TCP MUST be able to receive a TCP option in any segment"* and an example of the second kind is *"There MUST be a mechanism for reporting soft TCP error conditions to the application."* A TCP/IP implementation that violates requirements of the first kind may not be able to communicate with other TCP/IP implementations and may even lead to network failures. Violation of the second kind of requirements will only affect the communication within the system and will not affect host-to-host communication.

Table 1: TCP/IP features implemented by uIP and lwIP

Feature	uIP	lwIP
IP and TCP checksums	x	x
IP fragment reassembly	x	x
IP options		
Multiple interfaces		x
UDP		x
Multiple TCP connections	x	x
TCP options	x	x
Variable TCP MSS	x	x
RTT estimation	x	x
TCP flow control	x	x
Sliding TCP window		x
TCP congestion control	Not needed	x
Out-of-sequence TCP data		x
TCP urgent data	x	x
Data buffered for rexmit		x

In our implementations, we have implemented all RFC requirements that affect host-to-host communication. However, in order to reduce code size, we have removed certain mechanisms in the interface between the application and the stack, such as the soft error reporting mechanism and dynamically configurable type-of-service bits for TCP connections. Since there are only very few applications that make use of those features, we believe that they can be removed without loss of generality. Table 1 lists the features that uIP and lwIP implements.

5 Memory and buffer management

In our target architecture, RAM is the most scarce resource. With only a few kilobytes of RAM available for the TCP/IP stack to use, mechanisms used in traditional TCP/IP cannot be directly applied.

Because of the different design goals for the lwIP and the uIP implementations, we have chosen two different memory management solutions. The lwIP implementation has dynamic buffer and memory allocation mechanisms where memory for holding connection state and packets is dynamically allocated from a global pool of available memory blocks. Packets are contained in one or more dynamically allocated buffers of fixed size. The size of the packet buffers is determined by a configuration option at compile time. Buffers are allocated by the network device driver when an incoming packet arrives. If the packet is larger than one buffer, more buffers are allocated and the packet is split into the buffers. If the incoming packet is queued by higher layers of the stack or the application, a reference counter in the buffer is incremented. The buffer will not be deallocated until the reference count is zero.

The uIP stack does not use explicit dynamic memory allocation. Instead, it uses a single global buffer for holding packets and has a fixed table for holding connection state. The global packet buffer is large enough to contain one packet of maximum size. When a packet arrives from the network, the device driver places it in the global buffer and calls the TCP/IP stack. If the packet contains data, the TCP/IP stack will notify the corresponding application. Because the data in the buffer will be overwritten by the next incoming packet, the application will either have to act immediately on the data or copy the data into a secondary buffer for later processing. The packet buffer will not be overwritten by new packets before the application has processed the data. Packets that arrive when the application is processing the data must be queued, either by the network device or by the device driver. Most single-chip Ethernet controllers have on-chip buffers that are large enough to contain at least 4 maximum sized Ethernet frames. Devices that are handled by the processor, such as RS-232 ports, can copy incoming bytes to a separate buffer during application processing. If the buffers are full, the incoming packet is dropped. This will cause performance degradation, but only when multiple connections are running in parallel. This is because uIP advertises a very small receiver window, which means that only a single TCP segment will be in the network per connection.

Outgoing data is also handled differently because of the different buffer schemes. In lwIP, an application that wishes to send data passes the length and a pointer to the data to the TCP/IP stack as well as a flag which indicates whether the data is volatile or not. The TCP/IP stack allocates buffers of suitable size and, depending on the volatile flag, either copies the data into the buffers or references the data through pointers. The allocated buffers contain space for the TCP/IP stack to prepend the TCP/IP and link layer headers. After the headers are written, the stack passes the buffers to the network device driver. The buffers are not deallocated when the device driver is finished sending the data, but held on a retransmission queue. If the data is lost in the network and have to be retransmitted, the buffers on retransmission queue will be retransmitted. The buffers are not deallocated until the data is known to be received by the peer. If the connection is aborted because of an explicit request from the local application or a reset segment from the peer, the connection's buffers are deallocated.

In uIP, the same global packet buffer that is used for incoming packets is also used for the TCP/IP headers of outgoing data. If the application sends dynamic data, it may use the parts of the global packet buffer that are not used for headers as a temporary storage buffer. To send the data, the application passes a pointer to the data as well as the length of the data to the stack. The TCP/IP headers are written into the global buffer and once the headers have been produced, the device driver sends the headers and the application data out on the network. The data is not queued for retransmissions. Instead, the application will have to reproduce the data if a retransmission is necessary.

The total amount of memory usage for our implementations depends heavily on the applications of the particular device in which the implementations are to be run. The memory configuration determines both the amount of traffic the system should be able to handle and the maximum amount of simultaneous connections. A device that will be sending large e-mails while at the same time running a web server with highly dynamic web pages and multiple simultaneous clients, will require more RAM than a simple Telnet server. It is possible to run the uIP implementation with as little as 200 bytes of RAM, but such a configuration will provide extremely low throughput and will only allow a small number of simultaneous connections.

6 Application program interface

The Application Program Interface (API) defines the way the application program interacts with the TCP/IP stack. The most commonly used API for TCP/IP is the BSD socket API which is used in most Unix systems and has heavily influenced the Microsoft Windows WinSock API. Because the socket API uses stop-and-wait semantics, it requires support from an underlying multitasking operating system. Since the overhead of task management, context switching and allocation of stack space for the tasks might be too high in our target architecture, the BSD socket interface is not suitable for our purposes.

Instead, we have chosen an event driven interface where the application is invoked in response to certain events. Examples of such events are data arriving on a connection, an incoming connection request, or a poll request from the stack. The event based interface fits well in the event based structure used by operating systems such as TinyOS [10]. Furthermore, because the application is able to act on incoming data and connection requests as soon as the TCP/IP stack receives the packet, low response times can be achieved even in low-end systems.

7 Protocol implementations

The protocols in the TCP/IP protocol suite are designed in a layered fashion where each protocol performs a specific function and the interactions between the protocol layers are strictly defined. While the layered approach is a good way to design protocols, it is not always the best way to implement them. For the lwIP implementation, we have chosen a fully modular approach where each protocol implementation is kept fairly separate from the others. In the smaller uIP implementation, the protocol implementations are tightly coupled in order to save code space.

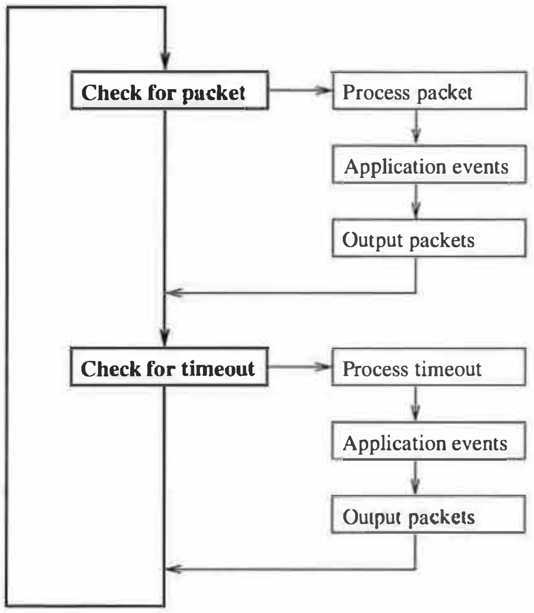


Figure 3: The main control loop.

7.1 Main control loop

The lwIP and uIP stacks can be run either as a task in a multitasking system, or as the main program in a singletasking system. In both cases, the main control loop (Figure 3) does two things repeatedly:

1. Check if a packet has arrived from the network.

2. Check if a periodic timeout has occurred.

If a packet has arrived, the input handler of the TCP/IP stack is invoked. The input handler function will never block, but will return at once. When it returns, the stack or the application for which the incoming packet was intended may have produced one or more reply packets which should be sent out. If so, the network device driver is called to send out these packets.

Periodic timeouts are used to drive TCP mechanisms that depend on timers, such as delayed acknowledgments, retransmissions and round-trip time estimations. When the main control loop infers that the periodic timer should fire, it invokes the timer handler of the TCP/IP stack. Because the TCP/IP stack may perform retransmissions when dealing with a timer event, the network device driver is called to send out the packets that may have been produced.

This is similar to how the BSD implementations drive the TCP/IP stack, but BSD uses software interrupts and a task scheduler to initiate input handlers and timers. In our limited system, we do not depend on such mechanisms being available.

7.2 IP — Internet Protocol

When incoming packets are processed by lwIP and uIP, the IP layer is the first protocol that examines the packet. The IP layer does a few simple checks such as if the destination IP address of the incoming packet matches any of the local IP address and verifies the IP header checksum. Since there are no IP options that are strictly required and because they are very uncommon, both lwIP and uIP drop any IP options in received packets.

7.2.1 IP fragment reassembly

In both lwIP and uIP, IP fragment reassembly is implemented using a separate buffer that holds the packet to be reassembled. An incoming fragment is copied into the right place in the buffer and a bit map is used to keep track of which fragments have been received. Because the first byte of an IP fragment is aligned on an 8-byte boundary, the bit map requires a small amount of memory. When all fragments have been reassembled, the resulting IP packet is passed to the transport layer. If all fragments have not been received within a specified time frame, the packet is dropped.

The current implementation only has a single buffer for holding packets to be reassembled, and therefore does not support simultaneous reassembly of more than one packet. Since fragmented packets are uncommon, we belive this to be a reasonable decision. Extending our implementation to support multiple buffers would be straightforward, however.

7.2.2 Broadcasts and multicasts

IP has the ability to broadcast and multicast packets on the local network. Such packets are addressed to special broadcast and multicast addresses. Broadcast is used heavily in many UDP based protocols such as the Microsoft Windows file-sharing SMB protocol. Multicast is primarily used in protocols used for multimedia distribution such as RTP. TCP is a point-to-point protocol and does not use broadcast or multicast packets.

Because lwIP supports applications using UDP, it has support for both sending and receiving broadcast and multicast packets. In contrast, uIP does not have UDP support and therefore handling of such packets has not been implemented.

7.3 ICMP — Internet Control Message Protocol

The ICMP protocol is used for reporting soft error conditions and for querying host parameters. Its main use is, however, the echo mechanism which is used by the `ping` program.

The ICMP implementations in lwIP and uIP are very simple as we have restricted them to only implement ICMP echo messages. Replies to echo messages are constructed by simply swapping the source and destination IP addresses of incoming echo requests and rewriting the ICMP header with the Echo-Reply message type. The ICMP checksum is adjusted using standard techniques [23].

Since only the ICMP echo message is implemented, there is no support for Path MTU discovery or ICMP redirect messages. Neither of these is strictly required for interoperability; they are performance enhancement mechanisms.

7.4 TCP — Transmission Control Protocol

The TCP implementations in lwIP and uIP are driven by incoming packets and timer events. IP calls TCP when a TCP packet arrives and the main control loop calls TCP periodically.

Incoming packets are parsed by TCP and if the packet contains data that is to be delivered to the application, the application is invoked by the means of a function call. If the incoming packet acknowledges previously sent data, the connection state is updated and the application is informed, allowing it to send out new data.

7.4.1 Listening connections

TCP allows a connection to listen for incoming connection requests. In our implementations, a listening connection is identified by the 16-bit port number and incoming connection requests are checked against the list of listening connections. This list of listening connections is dynamic and can be altered by the applications in the system.

7.4.2 Sending data

When sending data, an application will have to check the number of available bytes in the send window and adjust the number of bytes to send accordingly. The size of the send window is dictated by the memory configuration as well as the buffer space announced by the receiver of the data. If no buffer space is available, the application has to defer the send and wait until later.

Buffer space becomes available when an acknowledgment from the receiver of the data has been received.

The stack informs the application of this event, and the application may then repeat the sending procedure.

7.4.3 Sliding window

Most TCP implementations use a sliding window mechanism for sending data. Multiple data segments are sent in succession without waiting for an acknowledgment for each segment.

The sliding window algorithm uses a lot of 32-bit operations and because 32-bit arithmetic is fairly expensive on most 8-bit CPUs, uIP does not implement it. Also, uIP does not buffer sent packets and a sliding window implementation that does not buffer sent packets will have to be supported by a complex application layer. Instead, uIP allows only a single TCP segment per connection to be unacknowledged at any given time. lwIP, on the other hand, implements TCP's sliding window mechanism using output buffer queues and therefore does not add additional complexity to the application layer.

It is important to note that even though most TCP implementations use the sliding window algorithm, it is not required by the TCP specifications. Removing the sliding window mechanism does not affect interoperability in any way.

7.4.4 Round-trip time estimation

TCP continuously estimates the current Round-Trip Time (RTT) of every active connection in order to find a suitable value for the retransmission time-out.

We have implemented the RTT estimation using TCP's periodic timer. Each time the periodic timer fires, it increments a counter for each connection that has unacknowledged data in the network. When an acknowledgment is received, the current value of the counter is used as a sample of the RTT. The sample is used together with the standard TCP RTT estimation function [13] to calculate an estimate of the RTT. Karn's algorithm [14] is used to ensure that retransmissions do not skew the estimates.

7.4.5 Retransmissions

Retransmissions are driven by the periodic TCP timer. Every time the periodic timer is invoked, the retransmission timer for each connection is decremented. If the timer reaches zero, a retransmission should be made.

The actual retransmission operation is handled differently in uIP and in lwIP. lwIP maintains two output queues: one holds segments that have not yet been sent, the other holds segments that have been sent but not yet been acknowledged by the peer. When a retransmission is required, the first segment on the queue of segments that has not been acknowledged is sent. All other segments in the queue are moved to the queue with unsent segments.

As uIP does not keep track of packet contents after they have been sent by the device driver, uIP requires that the application takes an active part in performing the retransmission. When uIP decides that a segment should be retransmitted, it calls the application with a flag set indicating that a retransmission is required. The application checks the retransmission flag and produces the same data that was previously sent. From the application's standpoint, performing a retransmission is not different from how the data originally was sent. Therefore the application can be written in such a way that the same code is used both for sending data and retransmitting data. Also, it is important to note that even though the actual retransmission operation is carried out by the application, it is the responsibility of the stack to know when the retransmission should be made. Thus the complexity of the application does not necessarily increase because it takes an active part in doing retransmissions.

7.4.6 Flow control

The purpose of TCP's flow control mechanisms is to allow communication between hosts with wildly varying memory dimensions. In each TCP segment, the sender of the segment indicates its available buffer space. A TCP sender must not send more data than the buffer space indicated by the receiver.

In our implementations, the application cannot send more data than the receiving host can buffer. Before sending data, the application checks how many bytes it is allowed to send and does not send more data than the other host can accept. If the remote host cannot accept any data at all, the stack initiates the zero window probing mechanism.

The application is responsible for controlling the size of the window size indicated in sent segments. If the application must wait or buffer data, it can explicitly close the window so that the sender will not send data until the application is able to handle it.

7.4.7 Congestion control

The congestion control mechanisms limit the number of simultaneous TCP segments in the network. The algorithms used for congestion control [13] are designed to be simple to implement and require only a few lines of code.

Since uIP only handles one in-flight TCP segment per connection, the amount of simultaneous segments cannot be further limited, thus the congestion control mechanisms are not needed. lwIP has the ability to have multiple in-flight segments and therefore implements all of TCP's congestion control mechanisms.

7.4.8 Urgent data

TCP's urgent data mechanism provides an application-to-application notification mechanism, which can be used by an application to mark parts of the data stream as being more urgent than the normal stream. It is up to the receiving application to interpret the meaning of the urgent data.

In many TCP implementations, including the BSD implementation, the urgent data feature increases the complexity of the implementation because it requires an asynchronous notification mechanism in an otherwise synchronous API. As our implementations already use an asynchronous event based API, the implementation of the urgent data feature does not lead to increased complexity.

7.4.9 Connection state

Each TCP connection requires a certain amount of state information in the embedded device. Because the state information uses RAM, we have aimed towards minimizing the amount of state needed for each connection in our implementations.

The uIP implementation, which does not use the sliding window mechanism, requires far less state information than the lwIP implementation. The sliding window implementation requires that the connection state includes several 32-bit sequence numbers, not only for keeping track of the current sequence numbers of the connection, but also for remembering the sequence numbers of the last window updates. Furthermore, because lwIP is able to handle multiple local IP addresses, the connection state must include the local IP address. Finally, as lwIP maintains queues for outgoing segments, the memory for the queues is included in the connection state. This makes the state information needed for lwIP nearly 60 bytes larger than that of uIP which requires 30 bytes per connection.

8 Results

8.1 Performance limits

In TCP/IP implementations for high-end systems, processing time is dominated by the checksum calculation loop, the operation of copying packet data and context switching [15]. Operating systems for high-end systems often have multiple protection domains for protecting kernel data from user processes and user processes from each other. Because the TCP/IP stack is run in the kernel, data has to be copied between the kernel space and the address space of the user processes and a context switch has to be performed once the data has been copied. Performance can be enhanced by combining the copy operation with the checksum calculation [19]. Because high-end systems usually have numerous active connections, packet demultiplexing is also an expensive operation [17].

A small embedded device does not have the necessary processing power to have multiple protection domains and the power to run a multitasking operating system. Therefore there is no need to copy data between the TCP/IP stack and the application program. With an event based API there is no context switch between the TCP/IP stack and the applications.

In such limited systems, the TCP/IP processing overhead is dominated by the copying of packet data from the network device to host memory, and checksum calculation. Apart from the checksum calculation and copying, the TCP processing done for an incoming packet involves only updating a few counters and flags before handing the data over to the application. Thus an estimate of the CPU overhead of our TCP/IP implementations can be obtained by calculating the amount of CPU cycles needed for the checksum calculation and copying of a maximum sized packet.

8.2 The impact of delayed acknowledgments

Most TCP receivers implement the delayed acknowledgment algorithm [3] for reducing the number of pure acknowledgment packets sent. A TCP receiver using this algorithm will only send acknowledgments for every other received segment. If no segment is received within a specific time-frame, an acknowledgment is sent. The time-frame can be as high as 500 ms but typically is 200 ms.

A TCP sender such as uIP that only handles a single outstanding TCP segment will interact poorly with the delayed acknowledgment algorithm. Because the receiver only receives a single segment at a time, it will wait as much as 500 ms before an acknowledgment is sent. This means that the maximum possible throughput is severely limited by the 500 ms idle time.

Thus the maximum throughput equation when sending data from uIP will be $p = s/(t + t_d)$ where s is the segment size and t_d is the delayed acknowledgment timeout, which typically is between 200 and 500 ms. With a segment size of 1000 bytes, a round-trip time of 40 ms and a delayed acknowledgment timeout of 200 ms, the maximum throughput will be 4166 bytes per second. With the delayed acknowledgment algorithm disabled at the receiver, the maximum throughput would be 25000 bytes per second.

It should be noted, however, that since small systems running uIP are not very likely to have large amounts of data to send, the delayed acknowledgment throughput degradation of uIP need not be very severe. Small amounts of data sent by such a system will not span more than a single TCP segment, and would therefore not be affected by the throughput degradation anyway.

The maximum throughput when uIP acts as a receiver is not affected by the delayed acknowledgment throughput degradation.

8.3 Measurements

For our experiments we connected a 450 MHz Pentium III PC running FreeBSD 4.7 to an Ethernut board [16] through a dedicated 10 megabit/second Ethernet network. The Ethernut board is a commercially available embedded system equipped with a RealTek RTL8019AS Ethernet controller, an Atmel Atmega128 AVR microcontroller running at 14.7456 MHz with 128 kilobytes of flash ROM for code storage and 32 kilobytes of RAM. The FreeBSD host was configured to run the Dummynet delay emulator software [24] in order to facilitate controlled delays for the communication between the PC and the embedded system.

In the embedded system, a simple web server was run on top of the uIP and lwIP stacks. Using the `fetch` file retrieval utility, a file consisting of null bytes was downloaded ten times from the embedded system. The reported throughput was logged, and the mean throughput of the ten downloads was calculated. By redirecting file output to `/dev/null`, the file was immediately discarded by the FreeBSD host. The file size was 200 kilobytes for the uIP tests, and 200 megabytes for the lwIP tests. The size of the file made it impossible to keep it all in the memory of the embedded system. Instead, the file was generated by the web server as it was sent out on the network.

The total TCP/IP memory consumption in the embedded system was varied by changing the send window size. For uIP, the send window was varied between 50 bytes and the maximum possible value of 1450 bytes in steps of 50 bytes. The send window configuration translates into a total RAM usage of between 400 bytes and 3 kilobytes. The lwIP send window was varied between 500 and 11000 bytes in steps of 500 bytes, leading to a total RAM consumption of between 5 and 16 kilobytes.

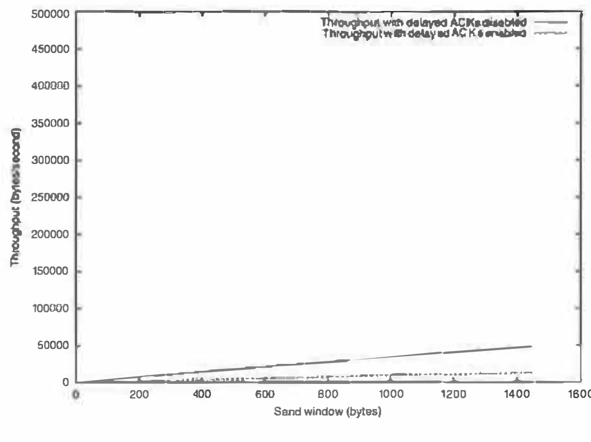


Figure 4: uIP sending data with 10 ms emulated delay.

Figure 4 shows the mean throughput of the ten file downloads from the web server running on top of uIP, with an additional 10 ms delay created by the Dummynet delay emulator. The two curves show the measured throughput with the delayed acknowledgment algorithm disabled and enabled at the receiving FreeBSD host, respectively. The performance degradation caused by the delayed ac-

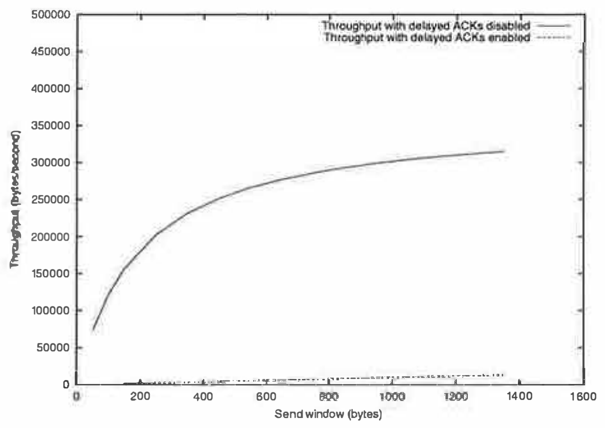


Figure 5: uIP sending data without emulated delay.

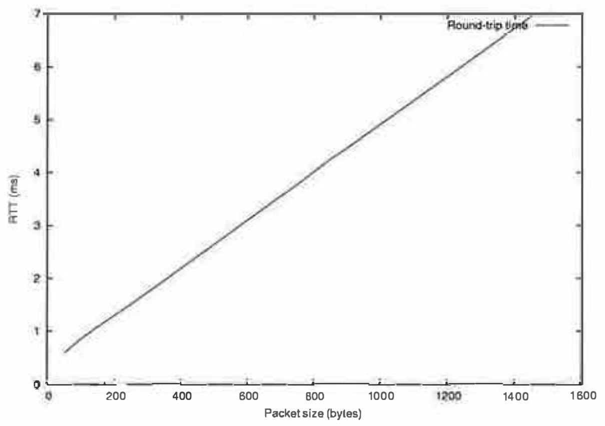


Figure 6: Round-trip time as a function of packet size.

knowledgments is evident.

Figure 5 shows the same setup, but without the 10 ms emulated delay. The lower curve, showing the throughput with delayed acknowledgments enabled, is very similar to the lower one in Figure 4. The upper curve, however, does not show the same linear relation as the previous figure, but shows an increasing throughput where the increase declines with increasing send window size. One explanation for the declining increase of throughput is that the round-trip time increases with the send window size because of the increased per-packet processing time. Figure 6 shows the round-trip time as a function of packet size. These measurements were taken using the `ping` program and therefore include the cost for the packet copying operation twice; once for packet input and once for packet output.

The throughput of lwIP shows slightly different char-

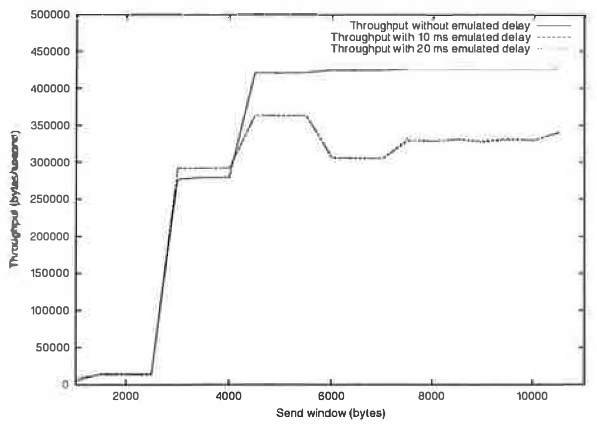


Figure 7: lwIP sending data with and without emulated delays.

acteristics. Figure 7 shows three measured throughput curves, without emulated delay, and with emulated delays of 10 ms and 20 ms. For all measurements, the delayed acknowledgment algorithm is enabled at the FreeBSD receiver. We see that for small send window sizes, lwIP also suffers from the delayed acknowledgment throughput degradation. With a send window larger than two maximum TCP segment sizes (3000 bytes), lwIP is able to send out two TCP segments per round-trip time and thereby avoids the delayed acknowledgments throughput degradation. Without emulated delay, the throughput quickly reaches a maximum of about 415 kilobytes per second. This limit is likely to be the processing limit of the lwIP code in the embedded system and therefore is the maximum possible throughput for lwIP in this particular system.

The maximum throughput with emulated delays is lower than without delay emulation, and the similarity of the two curves suggests that the throughput degradation could be caused by interaction with the Dummynet software.

8.4 Code size

The code was compiled for the 32-bit Intel x86 and the 8-bit Atmel AVR platforms using gcc [28] versions 2.95.3 and 3.3 respectively, with code size optimization turned on. The resulting size of the compiled code can be seen in Tables 2 to 5. Even though both implementations support ARP and SLIP and lwIP includes UDP, only the protocols discussed in this paper are presented. Because the protocol implementations in uIP are tightly coupled, the individual sizes of the implementations are

Table 2: Code size for uIP (x86)

Function	Code size (bytes)
Checksumming	464
IP, ICMP and TCP	4724
Total	5188

Table 3: Code size for uIP (AVR)

Function	Code size (bytes)
Checksumming	712
IP, ICMP and TCP	4452
Total	5164

not reported.

There are several reasons for the dramatic difference in code size between lwIP and uIP. In order to support the more complex and configurable TCP implementation, lwIP has significantly more complex buffer and memory management than uIP. Since lwIP can handle packets that span several buffers, the checksum calculation functions in lwIP are more complex than those in uIP. The support for dynamically changing network interfaces in lwIP also contributes to the size increase of the IP layer because the IP layer has to manage multiple local IP addresses. The IP layer in lwIP is further made larger by the fact that lwIP has support for UDP, which requires that the IP layer is able handle broadcast and multicast packets. Likewise, the ICMP implementation in lwIP has support for UDP error messages which have not been implemented in uIP.

The TCP implementation is lwIP is nearly twice as large as the full IP, ICMP and TCP implementation in uIP. The main reason for this is that lwIP implements the sliding window mechanism which requires a large amount of buffer and queue management functionality that is not required in uIP.

The different memory and buffer management schemes used by lwIP and uIP have implications on code size, mainly in 8-bit systems. Because uIP uses a global buffer for all incoming packets, the absolute memory addresses of the protocol header fields are known at compile time. Using this information, the compiler is able to generate code that uses absolute addressing, which on many 8-bit processors requires less code than indirect addressing.

Is it interesting to note that the size of the compiled lwIP

Table 4: Code size for lwIP (x86)

Function	Code size (bytes)
Memory management	2512
Checksumming	504
Network interfaces	364
IP	1624
ICMP	392
TCP	9192
Total	14588

Table 5: Code size for lwIP (AVR)

Function	Code size (bytes)
Memory management	3142
Checksumming	1116
Network interfaces	458
IP	2216
ICMP	594
TCP	14230
Total	21756

code is larger on the AVR than on the x86, while the uIP code is of about the same size on the two platforms. The main reason for this is that lwIP uses 32-bit arithmetic to a much larger degree than uIP and each 32-bit operation is compiled into a large number of machine code instructions.

9 Future work

Prioritized connections. It is advantageous to be able to prioritize certain connections such as Telnet connections for manual configuration of the device. Even in a system that is under heavy load from numerous clients, it should be possible to remotely control and configure the device. In order to do provide this, different connection types could be given different priority. For efficiency, such differentiation should be done as far down in the system as possible, preferably in the device driver.

Security aspects. When connecting systems to a network, or even to the global Internet, the security of the system is very important. Identifying levels of security and mechanisms for implementing security for embedded devices is crucial for connecting systems to the global Internet.

Address auto-configuration. If hundreds or even thousands of small embedded devices should be deployed, auto-configuration of IP addresses is advantageous. Such mechanisms already exist in IPv6, the next version of the Internet Protocol, and are currently being standardized for IPv4.

Improving throughput. The throughput degradation problem caused by the poor interaction with the delayed acknowledgment algorithm should be fixed. By increasing the maximum number of in-flight segments from one to two, the problem will not appear. When increasing the amount of in-flight segments, congestion control mechanisms will have to be employed. Those mechanisms are trivial, however, when the upper limit is two simultaneous segments.

Performance enhancing proxy. It might be possible to increase the performance of communication with the embedded devices through the use of a proxy situated near the devices. Such a proxy would have more memory than the devices and could assume responsibility for buffering data.

10 Summary and conclusions

We have shown that it is possible to fit a full scale TCP/IP implementation well within the limits of an 8-bit microcontroller, but that the throughput of such a small implementation will suffer. We have not removed any TCP/IP mechanisms in our implementations, but have full support for reassembly of IP fragments and urgent TCP data. Instead, we have minimized the interface between the TCP/IP stack and the application.

The maximum achievable throughput for our implementations is determined by the send window size that the TCP/IP stack has been configured to use. When sending data with uIP, the delayed ACK mechanism at the receiver lowers the maximum achievable throughput considerably. In many situations however, a limited system running uIP will not produce so much data that this will cause problems. lwIP is not affected by the delayed ACK throughput degradation when using a large enough send window.

11 Acknowledgments

Many thanks go to Martin Nilsson, who has provided encouragement and been a source of inspiration throughout the preparation of this paper. Thanks also go to Deborah Wallach for comments and suggestions, the anonymous reviewers whose comments were highly appreciated, and to all who have contributed bugfixes, patches and suggestions to the lwIP and uIP implementations.

References

[1] J. Bentham. *TCP/IP Lean: Web servers for embedded systems.* CMP Books, October 2000.

[2] R. Braden. Requirements for internet hosts – communication layers. RFC 1122, Internet Engineering Task Force, October 1989.

[3] D. D. Clark. Window and acknowledgement strategy in TCP. RFC 813, Internet Engineering Task Force, July 1982.

[4] G. H. Cooper. TinyTCP. Web page. 2002-10-14.
URL: *http://www.csonline.net/bpaddock/tinytcp/*

[5] Atmel Corporation. Embedded web server. AVR 460, January 2001. Avalible from www.atmel.com.

[6] A. Dannenberg. MSP430 internet connectivity. SLAA 137, November 2001. Avalible from www.ti.com.

[7] A. Dunkels. lwIP - a lightweight TCP/IP stack. Web page. 2002-10-14.
URL: *http://www.sics.se/~adam/lwip/*

[8] A. Dunkels. uIP - a TCP/IP stack for 8- and 16-bit microcontrollers. Web page. 2002-10-14.
URL: *http://dunkels.com/adam/uip/*

[9] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the internet. *IEEE/ACM Transactions on Networking*, August 1999.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[11] InterNiche Technologies Inc. NicheStack portable TCP/IP stack. Web page. 2002-10-14.
URL: *http://www.iniche.com/products/tcpip.htm*

[12] LiveDevices Inc. Embedinet - embedded internet software products. Web page. 2002-10-14.
URL: *http://www.livedevices.com/net_products/embedinet.shtml*

[13] V. Jacobson. Congestion avoidance and control. In *Proceedings of the SIGCOMM '88 Conference*, Stanford, California, August 1988.

[14] P. Karn and C. Partridge. Improving round-trip time estimates in reliablie transport protocols. In *Proceedings of the SIGCOMM '87 Conference*, Stowe, Vermont, August 1987.

[15] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of the ACM SIGCOMM '93 Symposium*, pages 259–268, September 1993.

[16] H. Kipp. Ethernut embedded ethernet. Web page. 2002-10-14.
URL: *http://www.ethernut.de/en/*

[17] P. E. McKenney and K. F. Dove. Efficient demultiplexing of incoming TCP packets. In *Proceedings of the SIGCOMM '92 Conference*, pages 269–279, Baltimore, Maryland, August 1992.

[18] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System.* Addison-Wesley, 1996.

[19] C. Partridge and S. Pink. A faster UDP. *IEEE/ACM Transactions in Networking*, 1(4):429–439, August 1993.

[20] J. Postel. Internet protocol. RFC 791, Internet Engineering Task Force, September 1981.

[21] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[22] Kadak Products. Kadak KwikNET TCP/IP stack. Web page. 2002-10-14.
URL: *http://www.kadak.com/html/kdkp1030.htm*

[23] A. Rijsinghani. Computation of the internet checksum via incremental update. RFC 1624, Internet Engineering Task Force, May 1994.

[24] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[25] C. Shannon, D. Moore, and K. Claffy. Beyond folklore: Observations on fragmented traffic. *IEEE/ACM Transactions on Networking*, 10(6), December 2002.

[26] H. Shrikumar. IPic - a match head sized webserver. Web page. 2002-10-14.
URL: *http://www-ccs.cs.umass.edu/~shri/iPic.html*

[27] CMX Systems. CMX-MicroNet true TCP/IP networking. Web page. 2002-10-14.
URL: *http://www.cmx.com/micronet.htm*

[28] The GCC Team. The GNU compiler collection. Web page. 2002-10-14.
URL: *http://gcc.gnu.org/*

System Services for Ad-Hoc Routing: Architecture, Implementation and Experiences

Vikas Kawadia*
ECE Dept and CSL, UIUC
kawadia@uiuc.edu

Yongguang Zhang
HRL Laboratories, LLC
ygz@hrl.com

Binita Gupta
Qualcomm Inc.
bgupta@qualcomm.com

Abstract

This work explores several system issues regarding the design and implementation of routing protocols for ad-hoc wireless networks. We examine the routing architecture in current operating systems and find it insufficient on several counts, especially for supporting on-demand or reactive routing protocols. Examples include lack of mechanisms for queuing outstanding packets awaiting route discovery and mechanisms for communicating route usage information from kernel to user-space. We propose an architecture and a generic API for any operating system to augment the current routing architecture. Implementing the API may normally require kernel modifications, but we provide an implementation for Linux using only the standard Linux 2.4 kernel facilities. The API is provided as a shared user-space library called the Ad-hoc Support Library (ASL), which uses a small loadable kernel module. To prove the viability of our framework, we provide a full-fledged implementation of the AODV protocol using ASL, and a design for the DSR protocol. Through this study, we also reinforce our belief that it is profoundly important to consider system issues in ad-hoc routing protocol design.

1 Introduction

Routing datagrams in a mobile ad-hoc network (MANET) is a difficult problem. Existing protocols to solve this problem include, but are not limited to AODV [25], DSR [20], DSDV [26] and TORA [24]. However most of the existing studies of these protocols are simulation based, with few real implementations. Validating MANET algorithms in real systems is necessary for their proliferation in the real world. But the sophisticated system-level programming so often required in the implementation of an ad-hoc routing protocol discourages MANET researchers from pursuing the much needed experimental studies.

*Part of this work was performed at HRL Laboratores, LLC when the author was a summer intern.

We believe the core reason for having such difficulties in implementation is the lack of system support and programming abstractions in general purpose operating systems (such as Unix/Linux). As we will explain later in this paper, ad-hoc routing protocols often employ new routing models or have special requirements that are not directly supported by the current operating systems. Without proper systems support and convenient programming abstractions, implementors are forced to do low-level system programming, and often end up making unplanned changes to the system internals in order to gain the additional functionality required for ad-hoc routing. Not only is this a non-trivial task, but in practice it can also lead to unstable systems, incompatible changes (by different implementations), and undeployable solutions.

To address these issues, we develop the system support and programming abstractions needed to facilitate MANET protocol implementations and deployment. Our solution provides a set of system services that provide the necessary system support to meet the requirements of most ad-hoc routing protocols. The new programming abstractions also allow easy programming of ad-hoc routing protocols without the need for low-level system programming.

In this paper, we explore the difficulties encountered in implementing MANET routing protocols in real operating systems, and study the common requirements imposed by MANET routing on the underlying operating system services. Then, we propose a general modification of the current IP routing architecture, and a specific implementation of this architecture in Linux. Finally, we present our experiences in implementing several MANET routing protocols under this framework.

2 Challenges in Mobile Ad-Hoc Routing

2.1 Current Routing Architecture

The current Internetworking architecture segregates the routing functionality into two parts: *packet forwarding* and *packet routing* (see Section 4.2 of Peterson & Davie's "Computer Networks" [28] for a good discussion on this topic). Packet forwarding refers to the process of taking a packet, consulting a table (the forwarding table), and sending the packet towards its destination as determined by that table. Packet routing, on the other hand, refers to the process of building the forwarding table. Forwarding is a well-defined process performed locally at each node, whereas routing involves a complex distributed decision making process commonly referred to as the routing algorithm or the routing protocol. The forwarding table contains enough information to accomplish the forwarding function, whereas the routing table contains information used by the routing algorithm to discover and maintain routes. Strictly speaking, these two tables are different data structures but the two terms are often used interchangeably.

In modern operating systems, packet forwarding is implemented inside the OS kernel whereas routing is implemented in the user space as a daemon program (the routing daemon). Figure 1 illustrates the general routing architecture. The forwarding table is inside the kernel and is often called the kernel routing table or route table. Whenever the kernel receives a packet, it consults this table, and forwards the packet to the "next-hop" neighbor through the corresponding network "interface". The kernel routing table is populated by the routing daemon according to the routing algorithm it implements.

There are numerous reasons for separating forwarding and routing [28], and placing packet forwarding inside the kernel and packet routing in user-space. Packet forwarding must make decisions for every packet and therefore should be efficient. It should reside inside the kernel so that a packet can traverse this node as fast as possible. On the other hand, packet routing involves complex and CPU/memory intensive tasks, which are properly situated outside the kernel. This principle of separation has made routing in modern operating systems efficient and flexible. It allows the routing function to continue evolving and improving without changing the OS kernel.

2.2 Challenges in On-demand Routing

It is desirable to fit mobile ad-hoc routing into the above architecture. Most ad-hoc routing protocols can be classified into two categories: proactive and reactive protocols. Pro-active (or table-driven) routing protocols maintain routes to all possible destinations by periodically exchanging control messages. Reactive (or on-demand) protocols, on the other hand, discover routes only when there is a demand for it. Proactive protocols (such as DSDV [26]) can be easily implemented as user-space routing daemons in the current routing architecture, in much the same way as routing protocols of the wired world (such as RIP, OSPF, or BGP). However, problems arise with reactive or on-demand routing protocols, such as AODV [25] and DSR [15]. We now describe these challenges in detail.

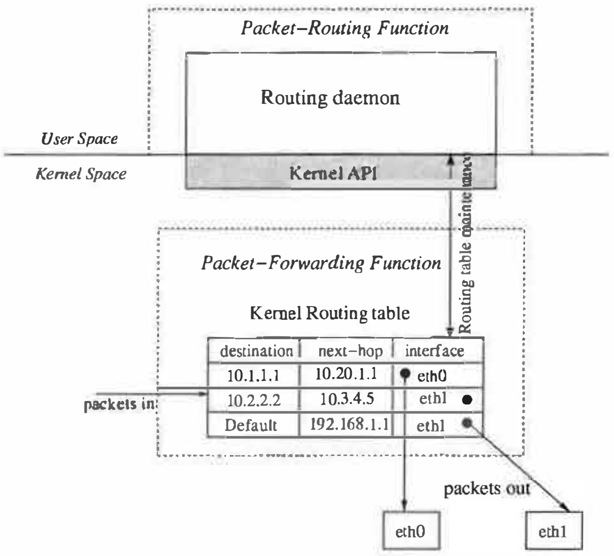


Figure 1. Current Routing Architecture

Challenge 1 *Handling Outstanding Packets*

Normally, each packet traversing the packet forwarding function will be matched against the kernel routing table. If no entry matches its destination, the kernel will drop the packet immediately. However, this is not a desirable behavior for on-demand ad-hoc routing. In on-demand ad-hoc routing, not all routes will exist *apriori*; some must be "discovered" when needed [15]. In such cases, the correct behavior should be: to notify the ad-hoc routing daemon of a route request, and to withhold the packet until the discovery finishes and route table updated. Unfortunately, there is no mechanism in modern operating systems to support this new packet forwarding behavior, and there is insufficient kernel support to implement tasks like queuing of all outstanding packets. Therefore, the operating system should implement the following functions for on-demand ad-hoc routing:

1. Identify the need for a route request.

2. Notify ad-hoc routing daemon of a route request.

3. Queue outstanding packets waiting for route discovery.

4. Re-inject them after successful route discovery.

Challenge 2 *Updating the Route Cache*

On-demand routing protocols typically maintain a cache of recently used routes in user space to optimize the route discovery overhead. Each entry in this route cache has an expiration timer, which needs to be reset when the corresponding route is used. The entry should be deleted (both from the user-space route cache and the kernel routing table) when the timer for that entry expires. Therefore, when an entry in the kernel routing table remains unused (i.e., has not been looked up) for a predefined time period, this information should be accessible to the the user-space routing daemon. This is difficult to achieve under the current routing architecture, because no record of route usage in the kernel is available to user-space programs.

Challenge 3 *Intermixing Forwarding and Routing Functions*

Certain ad-hoc routing protocols do not have a clean separation between the forwarding and routing functions in their design. Many of these protocols (notably DSR [15]) are based on the "on-demand behavior", where actions are taken only on reaction to data packets [21]. Since there is no periodic activities such as router advertisements, link/neighbor status sensing, or even the timely expirations of unused routing table or cache entries, routing activities must be made part of the forwarding activities. This protocol design presents a big implementation challenge to fit in the current routing architecture. There are two basic implementation approaches. The in-kernel approach typically requires the bulk of their routing logic to be implemented inside the kernel, making it difficult to program and to modify. On the other hand, the user-space approach requires the forwarding action to take place in user space, forcing every packet into user space.

In some cases, the principle of separation is violated for subtle optimizations aimed at reducing routing overhead. Such optimizations are usually simple to implement in a simulation environment, but present significant system design challenges. In the course of this study, we have found such examples in protocol design. We will present them in the later sections when we describe our experiences in implementing them.

Challenge 4 *New Routing Models*

Some ad-hoc routing protocols adopt unconventional routing models such as source routing ([15]), flow-based forwarding ([13]), etc. These new routing models deviate from the current IP routing architecture and present new challenges in system design. For example, in source routing the entire path that a packet should traverse is determined by the origin of the packet and encoded in the packet header, whereas in traditional IP routing the forwarding decision is made hop-by-hop and driven by the local routing tables. In flow-based forwarding each packet carries a flow id, and each node in the network has a flow table, which maintains the next-hop address and other information for each flow id. The forwarding is driven by table lookup on the flow id, and routing is the process to establish the flow table in each node.

Most general purpose operating systems are not flexible enough to provide a blanket support for all these and other new routing models. Implementation of these routing protocols will either modify the kernel IP stack to incorporate new routing architecture, or use kernel extension mechanisms (such as loadable modules) to bypass the IP stack.

Challenge 5 *Cross-Layer Interactions*

The wireless channel presents a lot of opportunities for optimizations through cross-layer interactions. For example, some ad-hoc routing protocols use physical and link layer parameters like signal strength, link status sensing, and link layer supported neighbor discovery, etc., in their routing algorithm. The problem of dealing with cross-layer interactions is a more difficult problem, both at the conceptual level and the implementation level. At the conceptual level, substantial work is needed in laying down guidelines for systematizing cross-layer interactions. This is necessary, since even though cross-layer design may provide optimizations, an indiscriminate access to all lower layer parameters would seriously damage the neat architecture which lies at the foundation of all networking.

At the implementation level, such cross-layer interactions obviously depend on the hardware capabilities and whether the hardware allows access to that information. It may be possible to provide access to lower layer parameters for some particular hardware, but a general solution for all hardware would require some standardization across the plethora of wireless interface cards and drivers available.

We believe that Challenges 1 and 2 can be met with enhanced system services in the operating systems. However, Challenge 3 and 4 may have to be dealt with on a case by case basis for every protocol. In this work, we first develop a general architecture to meet the first two challenges. And we will illustrate how we deal with the remaining challenges through our implementations of some ad-hoc routing protocols. Cross-layer interactions will also be the subject of future investigations.

3 New Architecture and API

We first develop a general solution to support on-demand routing in general purpose operating systems. The purpose is to suggest modifications to these operating systems so that ad-hoc routing can be easily supported in the future. We propose enhancements to the current packet-forwarding function with the following mechanisms.

An additional flag should be added to each kernel routing table entry to denote whether it is an *on-demand* entry, defined as those entries for which the kernel is prepared to queue packets in case of route unavailability. An on-demand route entry is said to be *deferred* if it has empty next-hop or interface fields, meaning that the route is yet to be discovered. Instead of getting dropped in the normal packet forwarding path, packets matching a deferred route will be processed as described in Challenge 1. We note that it is not necessary to include every possible on-demand destination in the routing table. Flagging a subnet-based route or the default route as on-demand can serve the same purpose.

A new component, called the on-demand routing component (ODRC), should be added to complement the kernel packet-forwarding function and implement the desired on-demand routing functionalities. When it receives a packet for a deferred route, it first notifies the user-space ad-hoc routing daemon of a *route request* for the packet's destination. Then, it stores the packet in a temporary buffer and waits for the ad-hoc routing daemon to return with the route discovery status. Once this process finishes and the corresponding kernel routing table entry is populated, the stored packets are removed from the temporary buffer and re-injected into packet forwarding.

To address Challenge 2, a timestamp field needs to be added to each route entry to record the last time this entry was used in packet forwarding. This timestamp can be used to retire a stale route that has not been used for a long time.

Finally, we provide a programming abstraction (API) so that these new mechanisms can be conveniently used in an ad-hoc routing daemon program. The API should contain the following functions:

-
  ```
  int route_add(addr_t dest,
          addr_t next_hop, char *dev);
  int route_del(addr_t dest);
  ```

  These basic routines add or delete an on-demand entry from kernel routing table. To add a deferred route entry, specify `next_hop` to be 0. (Here, `addr_t` is a generic type for the network address, such as `unsigned long` for IPv4 address.)

-
  ```
  int open_route_request();
  int read_route_request(int fd,
          struct route_info *r_info);
  ```

  ODRC notifies the ad-hoc routing daemon about the route requests in the form of an asynchronous stream. The first function returns a file descriptor for this stream and the second function fills in information about the route requests in the second argument, `struct route_info*` which is defined as follows :

  ```
  struct route_info {
          addr_t dest;
          addr_t src;
          u_int8_t protocol;
  };
  ```

  This structure contains information about the packet that triggers the route request, which can be useful for some routing daemons. For example a different action may be warranted if the packet was generated locally rather than being forwarded for some other host (which can be deduced from the `src` field). Similarly, some routing protocols may need to know if the route request is for a TCP packet or a UDP packet.

  The file descriptor semantics allows the ad-hoc routing daemon to use either event-driven or polling strategy. The function `read_route_request()` blocks until the next route request becomes available.

-
  ```
  int route_discovery_done(addr_t dest,
          int result);
  ```

  This function informs the ODRC that a route discovery for the given destination has finished and the kernel routing table populated. The result field indicates whether the route discovery was successful or not.

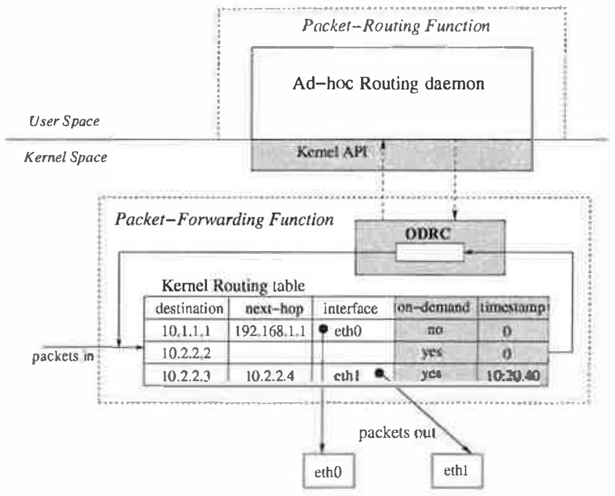
-
  ```
  int query_route_idle_time(addr_t dest);
  ```

  Given a destination, this function returns the idle time recorded in the kernel routing table for this entry (elapsed time since the last use of the route).

-
  ```
  int close_route_request(int fd);
  ```

  This function is called by the routing daemon when it no more desires to receive any more route requests. This enables the ODRC to free the memory used up by packets already queued up and close the fd.

Figure 2 illustrates our new architecture and its components. The shaded parts are our proposed additions.



destination	next–hop	interface	on–demand	timestamp
10.1.1.1	192.168.1.1	eth0	no	0
10.2.2.2			yes	0
10.2.2.3	10.2.2.4	eth1	yes	10:20.40

Figure 2. New Routing Architecture

The API we have provided takes care of Challenges 1 and 2. However, it is not yet complete as Challenges 3 and 4 have not yet been completely addressed.

Implementing this API in a Unix-like modern operating system usually requires some changes to the system internals. The ideal way is to integrate the above mechanisms into the kernel IP stack. This would involve implementing queuing for every deferred route and adding a special file descriptor for route-request stream. Depending on the operating system's kernel facilities and extensibility, this architecture can also be implemented as kernel modules, or largely in user-space.

It is debatable whether the ODRC functionality should be implemented outside the kernel and whether it is better to queue all deferred-routing packets in user-space. The advantage of a user-space approach is that it reduces the kernel complexity and memory usage. If the routing protocol requires prolonged route discovery procedure, it will be compelling to buffer packets outside the kernel. The disadvantage is the need to copy every deferred-routing packet (i.e., packets awaiting route discovery) from kernel to user-space and to re-inject them back to kernel when the routes are ready, but it can be argued that such overhead is insignificant compared with the time and overhead in an average route discovery.

4 Implementation in Linux: Ad-hoc Support Library

Our long-term objective is to implement this solution in common operating systems and make it standard in future versions. However, our immediate goal is to make it available to the current Linux 2.4 users because getting changes accepted into a standard operating system is a tedious process. We would like to find a way to provide the services described in Section 3 without being intrusive. This strategy certainly has practical value as few users are willing to modify their operating system kernels. To do this efficiently needs a careful design, which we describe in this section.

Linux provides several mechanisms for extending the kernel functionalities. These include loadable modules, where a new kernel function can be inserted into a running kernel without recompiling or rebooting, and a packet filtering and mangling facility called Netfilter [5]. In particular, Netfilter provides a set of hooks in the kernel networking stack where kernel modules can register callback functions, and allows them to mangle each packet traversing the corresponding hooks. We use these two mechanisms to implement our system services and make it our goal not to change the kernel source code.

4.1 Design and Mechanisms

We place the ODRC function in user-space to reduce the kernel complexity and memory requirements. There are two possible ways to implement this. One approach is to put ODRC in a shared library and link any routing daemon that wishes to use this ODRC function with this library. Another approach is to put ODRC in a separate daemon program and let it communicate with the routing daemon using some inter-process communication mechanism like sockets. Both approaches have their pros and cons. The library approach is more efficient because it does not have the overhead of inter-process communication, but any bug in the library is likely to crash the routing daemon also. The library approach gives a more natural picture of the ODRC functionalities as system services, i.e., the API is available as direct function calls once the appropriate header files are included.

We thus implement ODRC as a user-space library. We call it the Ad-hoc Support Library (ASL) or libASL. ASL implements the API we described in Section 3. The implementation consists of two main components: the first component is completely in user-space and implements the common functionalities which are needed by most on-demand routing daemons; the other part is specific to particular routing protocols and is implemented as loadable kernel modules. For example, for the AODV protocol there is the aodv-helper kernel module which provides additional API for some subtle optimizations prescribed by the AODV draft. The dsr-helper module is more complicated to accommodate for DSR's sophisticated features. We also provide a generic helper module called the route-check, which provides a simple solution to the route caching problem. This architecture consisting of libASL and helper modules provides ASL with the flexibility to incorporate future routing protocols or modifications to current ones in their respective helper modules.

4.1.1 Handling Outstanding Packets

To solve the problem of identifying the need for a route request, we need to filter all packets for which there exists no route. Without modifying the routing table structure, there is no simple way to do that in kernel. We solve this with an unused local tunnel device called Universal TUN/TAP (`tun`) as the "interface" device for these destinations. To catch packets for all such destinations, we can use the default route which is used for packets which do not match any other entry in the routing table. The default route can be setup like this :

```
ifconfig tun 127.0.0.2 netmask 255.255.255.255 \
        broadcast 127.0.0.2 up
route add default dev tun
```

TUN/TAP is a virtual tunnel device that makes available all received packets to a user-space program through the /dev/net/tun device. In our implementation, this device is opened by a call to `open_route_request()`, hence it receives all packets that kernel writes to `tun`, i.e., all packets for which there is no route. This also solves the problem of passing and storing packets in user-space.

Whenever a new packet is read from the virtual device, `/dev/net/tun`, the ad-hoc routing daemon which has opened a route request gets notified on that fd. It can read the details of the route request through `read_route_request()`, and then can initiate route discovery for the requested destinations. These packets are temporarily queued in a hash table keyed by the destination IP address. This functionality is implemented in the Ad-hoc Support Library. Since the buffer is in user-space, a large buffer is available to queue packets. This means that packets would not be lost even if the route discovery delays are large.

The next issue is to re-inject packets back into the IP stack after a successful route discovery. The mechanism we use is a raw IP socket[1]. A packet sent through a raw socket is inserted as is (bypassing any IP and header processing) to the kernel output chain just before the packet-forwarding function. Here, we use a raw socket to send the queued packets out. These packets are appropriately routed in the kernel using the newly discovered routes.

A natural question to ask is that why don't we re-inject the packets back into the IP stack by writing it on the user end of the same virtual interface used earlier, i.e., /dev/net/tun. To the kernel it appears as if a packet has been received on the tun virtual interface, and it can do the routing as if it were a normal incoming packet. This approach works fine for packets which a node forwards, but unfortunately does not work for packets generated locally by the node. Packets which are generated locally already pass through the IP output routines, and when re-injected through tun appear on the forwarding chain. The forwarding chain does not allow packets in which the source IP address matches the local IP address, since this is an indication that the node's IP address is being spoofed by somebody else. Hence, we have to resort to raw IP sockets as described above.

[1]Raw sockets are normally used to handle packets that the kernel does not support explicitly. The `ping` program, for example, uses raw sockets to generate ICMP packets.

4.1.2 Updating the Route Cache

Now we come to Challenge 2, to refresh entries in the user-space route cache when a route is used in the kernel. Since we are not making changes to the kernel routing table, the only way is to maintain a separate timestamp table for each entry in the routing table. We thus design a simple kernel module called `route_check` to maintain this table and register it at Netfilter's `POST_ROUTING` hook (after routing table lookup and before entering the physical network interface). This means that every outgoing packet will pass through this module. It simply peeks at the packet header and updates the corresponding timestamp value. This timestamp information is made available to user-space programs using an entry in the `/proc` file system. The `query_route_idle_time()` function exposed by the ASL API reads this file (`/proc/asl/route_check`) to determine the idle time for a destination. The routing daemon can check the freshness of a route by reading this file, and delete the stale routes from the kernel routing table accordingly. The route_check module is a generic helper module, which is available to all the routing daemons.

Actually, the current Linux kernel does maintain a cache of most frequently used routes to make routing lookups efficient. When a route is first used it is looked up from the Forwarding Information Base (FIB) which is a complex data structure maintaining all the routes. After first use this entry is inserted in the route cache for fast lookup. It expires from the cache if not used for some length of time. Information about this route cache is exposed through the files `/proc/net/rt_cache` and `/proc/net/rt_cache_stat`. Unfortunately these files do not include information about the `last_use_time` of the entries. It is a very simple modification to the Linux kernel to make it output this information, but since we are not making any changes at all in the core kernel source as it would require kernel recompilation, we have adopted the `route_check` module approach just described. We emphasize that a very small change in the Linux kernel would make the route_check

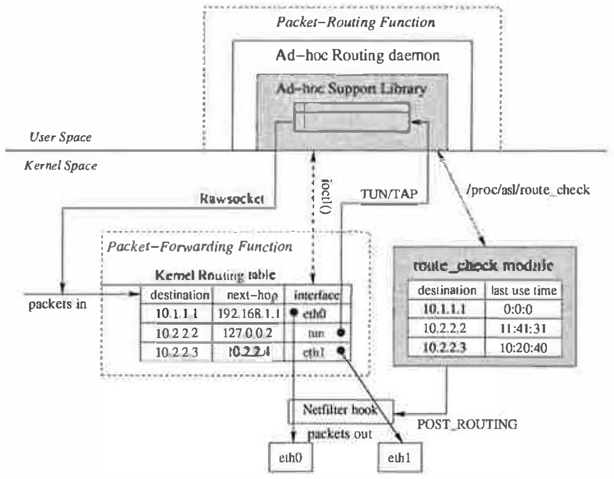


Figure 3. ASL software architecture

module unnecessary.

4.2 ASL Implementation Details

Figure 3 illustrates the structure of this implementation. The two main components are the user-space library ASL and the kernel module `route_check`. The library implements the API described in Section 3. We now describe how we implement these functions in our library.

`route_add()` and `route_del()` functions add or delete routes to the kernel using the `ioctl()` interface. When the user indicates that the route be a deferred route by specifying an empty next-hop, the device for the route is made to be `tun`. `open_route_request()` initializes the tun device, the raw socket, the data structures to queue deferred packets, and also inserts the `route_check` module in the kernel. The data structure to store the packets is a hash table of queues, keyed by the destination IP address. The function `open_route_request()` returns the descriptor of the tun device which can be monitored using a polling or event driven strategy. `read_route_request()` blocks reading from this tun device. When a packet is received on tun, this functions stores the packet and delivers information about the packet in the form of `struct route_info`. Based on this the routing daemon initiates route discovery, and calls the `route_discovery_done()` function on completion of this process. If the route discovery was successful then this function retrieves the packets for that destination from the storage and sends them out on the raw socket. If a route could not be found then the packets are thrown away and the memory used for them is freed. `query_route_idle_time()` reads the `last_use_time` for that destination from `/proc/asl/route_check` and returns the idle time. This needs to be called whenever the routing daemon has to make a decision to expire routes from its user-space route cache. The function `close_route_request()` simply shuts down all the sockets, frees all the memory for storing the packets, and removes the `route_check` module from the kernel.

Below we give the pseudo code of an example routing daemon which uses this library.

```
aslfd = open_route_request()
route_add(default,0) /* add deferred route */
loop  /* this could be select or poll */
  wait for input from {aslfd or other fd's}
  if input from aslfd
    dest = read_route_request()
    if(route request is new)
      do route discovery for dest
      if successful
        add route for dest to kernel
        route_discovery_done(success)
      else
        route_discovery_done(failure)
      end
    else
      continue
    end
  end
  if input from other fd's
    process according to protocol semantics
    /*call before expiring routes*/
    query_route_idle_time()
  end
end
close_route_request()
```

5 Implementing Routing Protocols: Experiences using ASL

To evaluate the utility of the Ad-hoc Support Library, we set out implementing the various routing protocols that have been proposed. We provide a full-fledged implementation of the AODV protocol. We also provide implementation design guidelines for some other protocols.

5.1 Ad-hoc On-demand Distance Vector Routing (AODV)

Our implementation of AODV is a user-space routing daemon which uses the Ad-hoc Support Library for system services. It follows a modular architecture in C++ to provide a clean and extensible implementation. The current implementation supports all the features of AODV draft version 10 [27]. In the following sections we describe both the user-space design as well as the special system support (in addition to standard ASL) required for AODV, which we implemented as the aodv-helper kernel module.

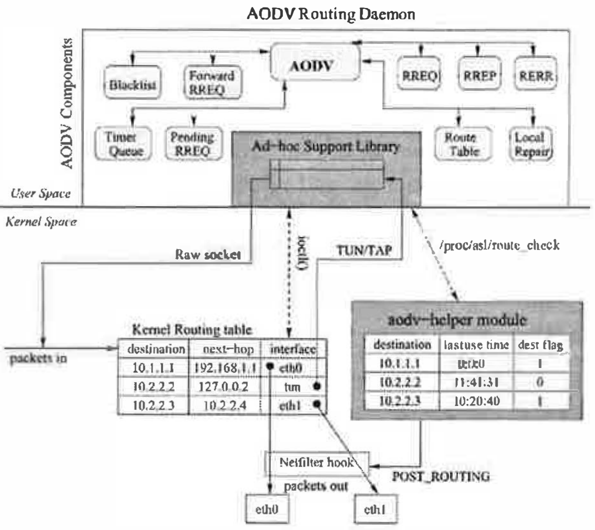


Figure 4. Software architecture of the AODV-UIUC routing daemon.

5.1.1 AODV Components

This section talks about the different components of our AODV routing daemon, their functionalities and the interactions among these components to implement various features of the AODV protocol. This is illustrated in Figure 4. Details of this implementation, called AODV-UIUC, are provided in [11].

The component called AODV defines the main flow of control inside the AODV routing daemon. The control flow is based on an event-driven design. The set of possible events include reception of routing control packets, expiration of various timers, and reception of route requests on the ASL socket. Possible actions include sending out packets, setting new timers and updating various data structures. The daemon program is essentially a big select() loop which monitors various file descriptors for the events and takes the appropriate actions. This component also initializes ASL by calling the functions `int route_add()` and `open\_route\_request()`.

The RREQ, RREP and RERR components take care of both generating as well as processing incoming route requests, route replies and route error packets respectively. The Routing Table component (routeTable) handles updates to the aodv routing table as well as to the kernel routing table. It also maintains a route cache using the aodv-helper module through the corresponding API function `query_route_idle_time_aodv()`, as explained in the next subsection. The Pending Route Request component (rreqPendingList) implements the expanding ring search and RREQ retransmission features of the AODV routing protocol. The Forward Route Request component ensures that a node does not process a particular RREQ packet multiple times, by storing a list of recently seen RREQ packets. The Local Repair component attempts to repair links locally and the BlackList component takes care of routing in the presence of unidirectional links. Finally, the TimerQueue component maintains various AODV timers including reboot timer, periodic refresh timer, hello timer and rreq retransmission timer.

5.1.2 ASL and Aodv-helper

Using ASL makes efficient on-demand routing possible in our AODV implementation. The generic route-check module can be used for maintaining the user-space route cache. However, the AODV protocol requires that whenever a packet is forwarded to any destination by a node using a particular route, the node should update the lifetime values (in its route-cache) associated with the destination, the previous hop and the next hop nodes on that route. Previous hop is defined as the next-hop along the reverse path back to the source. Updating the previous hop node, when a route is used was not possible using the generic route-check module, since the information about the previous hop is not available in the packet but only in the routing table . We had to redesign our data structures and the query process for updating the lifetime of the previous hop for a route. These substantially more complicated new data structures and query process were made part of the aodv-helper module.

Like the generic route_check module, the aodv-helper module also registers at the Netfilter's POST_ROUTING hook and peeks at the every outgoing packet to log in the timestamp information. But, unlike the route-check module, the aodv-helper module also logs an additional flag parameter called the destination flag with every entry in the /proc file. A value of 1 for this flag signifies that the entry correspond to the destination of the packet, and a value of 0 implies that the entry corresponds to the source of the packet. Thus, the aodv-helper module logs two entries for every outgoing packet, one for the destination and one for the source.

The `query_route_idle_time()` API function interface has also been modified as follows :

```
    int query_route_idle_time_aodv(addr_t k,
                    int destination_flag);
```

Given a destination k, this function returns the idle time recorded in the kernel routing table for this entry (elapsed time since the last use of the route). The destination_flag is used to differentiate between the return value. A value of 1, for the flag implies that the query is for the idle time since a packet was last forwarded to this destination, whereas a value of 0 denotes a query for the idle time since a packet was last received from this source. The aodv daemon, uses this API as follows:

1. Whenever the routing table entry for a destination d expires, the AODV routing daemon queries the aodv-helper module for the idle time for that destination with the following API function call (note the destination flag passed is 1): `query_route_idle_time_aodv(d,1)`.

2. If the destination d is just one hop away, then the routing daemon goes through the entire routing table looking for the nodes for which this destination acts as the next hop. It then queries aodv-helper module for the idle time corresponding to each of these nodes (with the destination flag set to both 1 and 0) and chooses the minimum idle time of all such idle times. The idle time for previous hops can be determined by a call to `query_route_idle_time()` API function with the destination flag set to 0, when the timer for such an entry expires. This ensures compliance with all the features of the draft.

5.1.3 Experiences

We found that it was pretty straightforward to implement AODV as a user-space daemon, once all the kernel interaction issues were taken care of by the Ad-hoc Support Library. The user-space daemon is about 5000 lines of C++ code and the aodv-helper module is about 800 lines. ASL itself is about 2500 lines of C code. Using ASL, AODV development was easier as all the debugging was confined to user-space. We have tested our implementation (See [11]) on a testbed of about 10 laptops.

We also realized that some of the subtle optimizations which needed hard work were probably not very important. For example, after implementing the aodv-helper, we realized that updating the previous hop and next hop lifetimes, when a route is used, is probably completely redundant. This is because the draft also says that nodes should maintain connectivity information with all one-hop neighbors, using either explicit hello messages, link layer notification mechanism or passive acknowledgments, and update the lifetime field for all the one-hop neighbors if the link is determined to be up. Since the set of possible previous hops and next hops is a subset of the set of one-hop neighbors, the lifetime for all such hops is automatically updated by the neighbor detection mechanism. Thus, updating next-hop and prev-hop lifetimes during route caching is probably unnecessary, as it comes into play only on those very rare occasions when a few consecutive hello packets from a neighbor were lost, but that neighbor was somehow still used as a previous/next hop on some route. Thus, if we ignore this redundant feature in the draft, aodv-helper is no more needed and the generic route-check module will be sufficient.

5.2 Dynamic Source Routing (DSR)

Our second attempt is to implement DSR within the ASL framework. We choose DSR because it is another popular and maturing ad-hoc routing protocol with significant research backing, and is also architecturally, a different protocol from AODV. The implementation starts with the DSR Internet Draft [14].

5.2.1 Difficulties

Implementing DSR within ASL framework is a big challenge. First, it is a source routing protocol and has its own protocol format to specify source routes (Challenge 4). Second, it is based entirely on on-demand behavior and does not have a separable routing and forwarding function (Challenge 3). Neither of these features fits directly in the ASL architecture.

In particular, we consider the following issues:

- *Interfacing with the kernel's IP stack.* DSR specifies its own protocol header, which is immediately after the IP header and before any IP payload. This implies that the naturally appropriate place for inserting the DSR implementation will be in the IP stack at the IP multiplex/demultiplexing point. There are other alternatives, such as intercepting every DSR packets at the packet input/output chains and bypassing the kernel IP stack, or processing the DSR packets in a virtual device driver below IP.

- *Processing every packet.* Every data packet carries the source route in the DSR header. As part of the packet forwarding process, this header information needs to be looked up and modified at each intermediate node. Further, the source route is also used to update the forwarding node's route cache. Route shortening in forms of gratuitous route reply may be applied if the forwarding note has a better route to the destination in its route cache. In addition, DSR control information (e.g., gratuitous route error messages) may be piggybacked on any packet, to reduce routing overhead. Therefore, all packets require significant processing. This makes fast packet forwarding difficult.

- *Maintaining routes* In the absence of periodic neighbor/link sensing, DSR relies on data packets to detect broken links. It requires every packet forwarded by a node to be acknowledged by the next hop, either through the possible built-in link-layer acknowledgment mechanism (such as 802.11 MAC), or by passive acknowledgment where the sending node overhears the next hop further forwarding the packet, or by explicit network layer acknowledgment from the next-hop back to the sender. This requires each node to keep a copy of all forwarded packets for a short period of time until being acknowledged. Packets unacknowledged after timeout period will trigger route error messages, and optionally salvaging actions where a forwarding node rewrites the packet's source route option to choose a different path. Route maintenance further complicates the system design.

- *Listening in the promiscuous mode.* DSR allows the optional use of promiscuous mode listening for performance improvement. Promiscuous listening is defined as the process by which the network card can overhear packets not intended for its hardware address, and deliver it to the network stack. Using this feature, a node can overhear a data packet and can add the source route to its route cache. Promiscuous mode requires support from hardware and device driver, but not all wireless devices supports this type of operation for security considerations.

5.2.2 A Split Design

Our goal, in accordance with the philosophy of this work, is to do a reasonably simple and maintainable implementation of DSR in Linux, with minimum modifications to the kernel source. As we have explained earlier, due to the inseparable forwarding and routing functions, there are usually two ways to implement such protocols: a complete in-kernel approach (such as in [19]), and a complete user-space approach (such as in [10]). Both approach have pros and cons. A complete user-space approach will be inefficient for the forwarding function, but an in-kernel approach is different to maintain, different to modify, and different to port to other operating systems.

In our implementation, we attempt a split-system approach. The idea is to segregate the forwarding and routing functions to some extent, even though they are intermixed in the protocol design (Challenge 3). We believe that the core of the source-routing based forwarding activities, i.e., to send a data packet to the next-hop based on its DSR header, should be as efficient as possible and reside inside the kernel. We call this the *source forwarding* function. The majority of other source routing activities, which are induced by source forwarding, need to be flexible and can reside in user-space.

Figure 5 illustrates the overall design of this DSR implementation; the shaded parts indicate the various DSR components. It consists of a user-space DSR Routing Daemon and two kernel modules: `DSR-forwarding-helper` and `DSR-maintenance-helper`. The user-space daemon performs majority of the DSR routing functions, including route discovery and route maintenance. It relies on ASL to manage on-demand route

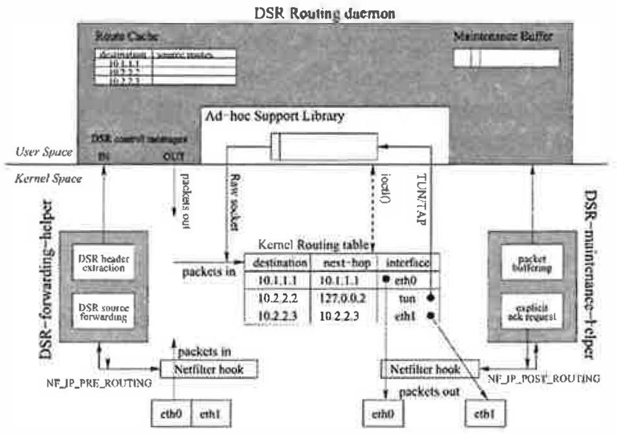


Figure 5. Design of DSR Routing Daemon

requests, and relies on the two kernel modules to interact with the forwarding function.

The `DSR-forwarding-helper` module handles all incoming DSR packets from the network devices. If it receives a DSR packet with a source route option, it executes the forwarding function, which involves making changes to the IP and the DSR headers. At the same time, it also extracts necessary DSR header information into a buffer. Later, this header information is passed on upward to the user-space route daemon and processed in the background, after the in-kernel forwarding is done. If the DSR packet is meant for this node, it is demultiplexed here (with the DSR header removed). Or, if it is a Route Request packet, it is sent upward to the DSR route daemon for source routing functions.

The `DSR-maintenance-helper` module inspects all outgoing DSR packets before sending to the network devices. Its only purpose is for route maintenance. It makes a copy of every outgoing packet and sends them upward to DSR route daemon for temporary buffering (in DSR Maintenance Buffer). Optionally, if the DSR route daemon determines that an explicit acknowledgment should be used, this module can insert a DSR Acknowledgment Request option in the DSR header of selected packets. Other than this, the route maintenance is almost entirely handled in user-space. The reason for this design is the following. We believe that route maintenance is not part of the core source forwarding function and should not stand in the way inside kernel. Once the packets are sent and copies are made, the route maintenance can work "in the background". When the acknowledgment comes back in the form of a DSR Acknowledgment option, the `DSR-forwarding-helper` module will forward it to the DSR route daemon, where the matching is done. If an entry in the Maintenance Buffer times out, the route daemon can update the Route Cache and generates Route

Error messages.

Under this design, a DSR data packet can pass through the kernel quickly without being delayed by non-critical DSR activities. Majority of other DSR activities are performed in user-space without getting in the way of the kernel source forwarding path. The kernel routing table will only contain entries for its neighbors. All other nodes will be marked as deferred (i.e., use `tun0`) so ASL can catch all the outstanding packets. ASL's `route_check` module is not used because DSR does not require periodic deletion of unused route cache entries.

5.2.3 Implementation

Following the same philosophy of the ASL work, our DSR implementation uses the standard Linux 2.4 kernel facilities only. All the kernel additions are implemented in two loadable kernel modules. No kernel recompilation is required.

We use the Netfilter facility extensively. The `DSR-forwarding-helper` module attaches itself to the Netfilter `NF_IP_PRE_ROUTING` hook to capture all incoming DSR packets from the network devices. The `DSR-maintenance helper` module attaches itself to the Netfilter `NF_IP_POST_ROUTING` hook to capture all outgoing DSR packets before passing to the network devices.

Our experience shows that an intermixing routing/forwarding protocol like DSR is more difficult to implement in a modern operating system, even with the help from the ASL framework. To achieve both efficiency and portability in the system design, we have to excise the routing/forwarding functional separation to some extent. Compared with the prior approaches (either all-in-kernel as in [19] or all-in-user-space as in [10]), our split design is better than the all-in-user-space approach because we now copy only the DSR header information, not the entire packet, to the user-space. The forwarding is entirely in kernel, while this header information can be processed later in the background, after the forwarding is done. This ensures a high performance forwarding function for DSR. Further, this design is better than the all-in-kernel approach, because kernel now process only the most critical function, not rest of the tedious on-demand behavior logics. The user-space implementation of the non-critical functions ensure that it is portable, maintainable, and extensible.

5.3 Other On-demand Routing Protocols

Temporally ordered routing algorithm (TORA) [24] is an adaptive, distributed routing algorithm based on the concept of link reversal. It strives to minimize communication overhead due to network topological changes. The TORA protocol specifications [23] are very amiable to the framework we have developed in this work. The algorithmic details can be implemented in user-space as the the TORA daemon, which uses the Ad-hoc Support Library for the on-demand mode of operation. For route caching, the generic route-check module seems sufficient.

Associativity based routing [30], is an on-demand, distance-vector routing protocol in which the metric is link-stability instead of the traditional hop-count based metric as in AODV. The link-stability is determined by associativity ticks which is essentially a count of beacons received from the neighbors. Since ABR specifies lots of features [31], which depend critically on the granularity of the associativity metric, it assumes that the data link layer is capable of getting a reactive estimate of the associativity metric efficiently. If the network card or the driver does not support this, ABR is not practicable for those devices. If such support is available, then ABR is quite cleanly implementable using our framework.

5.4 Pro-active Routing Protocols

Pro-active routing protocols can be easily implemented with the current routing architecture in all operating systems. They do not need the framework which we present. The DSDV (Destination Sequenced Distance Vector) routing protocol and the Adaptive DSDV protocol have been implemented in [11]. Another pro-active protocol called VDBP (Virtual Dynamic Backbone Routing) [18] has also been implemented for Linux. Optimized Link State Routing (OLSR) and Topology Broadcast with Reverse Path Forwarding (TBRPF) are two other popular proactive routing protocols which have also been successfully implemented in Linux and FreeBSD respectively.

Hybrid routing protocols use a combination of proactive and reactive routing schemes. For example, the Zone Routing Protocol (ZRP [12]) divides the network dynamically into zones, and uses a pro-active protocol for intra-zone routing whereas a reactive protocol for inter-zone routing. From a systems viewpoint, ZRP does not present any new challenges compared to AODV; ASL can be used for efficiently implementing the inter-zone routing part of ZRP.

6 Existing Implementations and Related Work

There have been several implementations of some on-demand ad-hoc routing protocols. These implementations address some or all of the on-demand routing problems, but very few attempt to provide a general framework as we do. In this section, we provide a comparison on how these implementations attempt to address the problems we described in Section 2, and suggest how our approach can help improving them.

An implementation study of AODV routing protocol [29] raises issues similar to what we have discussed here. To address on-demand routing problems for AODV, it suggests significant modifications to the existing kernel code. First, IP layer builds a short lived dummy routing table entry for every unroutable destination. It then uses netlink socket to inform AODV routing daemon about the need to initiate a route discovery. Data packets are buffered inside the kernel in a simple linked list referenced from the dummy routing table entry. IP is also modified to add a Last Use field for every route, which is used by the routing daemon when deleting the routes. Our independent investigations led to the identification of similar issues and development of the API we presented in Section 3. However instead of modifying the Linux kernel, we focused on providing a user-space implementation in the form of a shared library, which we hope will be of more immediate use in implementing ad-hoc routing protocols.

Madhoc is a user-space implementation of AODV [25]. To address the on-demand routing problem, it snoops ARP (Address Resolution Protocol) packets and uses them as an indication that the destination has no route and route discovery should be triggered. This scheme has a few serious drawbacks. First, the kernel generates an ARP request only if the destination belongs to the subnet of one of the network interfaces, or a host-specific route entry exists for this destination. This limits the applications to certain types of network configurations. Secondly, ARP will time out in relatively short time, and mad-hoc provides no mechanism to queue outstanding packets. This means that these packets might be dropped before the route discovery can be completed. Finally, ARP cache has a time-out value and snooping on ARP requests can result in spurious route requests when a next-hop node has been timed out in the ARP cache but the route is still valid.

AODV-UU [2] and AODV-UCSB [1] are two implementations of the AODV routing protocol. The kernel interaction part of the two implementations is the same. They differ only in the AODV protocol logic implementation, which is done in user-space. The kernel part consists of two Linux kernel modules (`kaodv` and `ip_queue_aodv`). To address the on-demand routing problems, these implementations use Netfilter to copy all packets from the kernel space to user-space. `kaodv` uses Netfilter to collect all packets before they enter the packet-forwarding function and `ip_queue_aodv` queues them to user-space. By matching these packets against the entries in the user-space route cache, packets for which there is no route can be identified and route request initiated. There are two obvious drawbacks of this approach: every packet has to cross the user kernel address space twice, inducing much overhead, and for every packet the routing is done twice as well, once in user-space and once again in the kernel. In our AODV-UIUC implementation, all such system interactions are cleanly taken care of by the Ad-hoc Support Library. The overhead of processing every packet in user-space is eliminated. The result is a much simpler, cleaner, and more efficient implementation.

Kernel-AODV [4] is another AODV implementation by NIST. The entire implementation is in the form of Linux kernel modules. As we have discussed earlier in Section 2, it is not a good system design to put the entire routing protocol in kernel-space. The complex protocol processing can slow the kernel, hog the memory, and crash the whole system if there are any bugs in the routing protocol.

Another kernel implementation of AODV [9] has been done by extending ARP. Note that unlike Mad-hoc which uses the ARP mechanism just to detect route requests, this approach reuses the ARP code in the kernel to do a complete implementation of the AODV protocol. Modified ARP requests and replies are used to generate AODV RREQ and RREP packets. ARP table essentially acts as the AODV routing table. Modified ARP reply with a special flag is used to generate RERR messages. Data packets are buffered inside the kernel in an ARP queue. This approach is a smart idea, but has the limitations of an in-kernel implementations which we have discussed above.

Dynamic Source Routing (DSR) [15, 19] has been implemented by the Rice University Monarch project in FreeBSD [3]. This implementation is entirely in-kernel and does extensive modifications in the kernel IP stack. While the implementation is a commendable project, it is difficult to maintain and update due to its complexity. For example, this implementation of DSR was developed on FreeBSD 2.2.7 and has only been upgraded to FreeBSD 3.3, whereas the current version of FreeBSD is 4.6.2 when this paper is written. Porting it to other operating systems will be prohibitively difficult.

Internet Manet Encapsulation Layer (IMEP) [8] is an encapsulation protocol proposed for manet routing, which provides certain common functions like single-interface abstraction, link status sensing, control message aggregation, and reliable broadcasting. IMEP attempts to provide a unified framework for all other ad-hoc routing protocols at the protocol processing level, whereas we attempt to provide a common module and a common interface at the implementation level. These two approaches are complementary. A simple user-space implementation of IMEP is possible using our framework. Additional support may be needed from the network device driver for link status sensing.

Temporally ordered link reversal algorithm (TORA) [24] has been implemented in Linux by University of Maryland [6]. It has been implemented over IMEP and hence can benefit from our framework. TORA can independently be implemented over our framework too.

University of Colorado has implemented several routing protocols [32, 22, 10] using MIT's Click Modular Router [17]. Click is a software architecture for building flexible and configurable routers. It provides basic protocol processing modules called elements, each of which implements a specific function like packet classification, queuing, and scheduling. Protocol developers write Click configuration files to instantiate elements and to connect them in a way to implement the protocol. In this aspect, Click can be a good candidate to meet Challenge 4. Nevertheless, our work focuses on common operating systems, whereas Click is designed for a special purpose application (fast and configurable routers).

A recent work [16] aims to provide a library of utilities for manet routing protocols, much like the GNU Zebra project does for wired routing protocols. It plans to provide utilities for timer management, neighbor discovery, managing tables etc. It however does not systematically deal with all the system issues, as we have done. ASL could be used in building this framework.

MagnetOS [7] is a distributed, power-aware, adaptive operating system which abstracts an ad-hoc network as a unified Java Virtual Machine. It allows for objects and components to automatically migrate among nodes in the network to optimize system performance. MagnetOS focuses on sensor networks, using distributed operating system techniques.

7 Conclusions

In this work, we study the operating system services for mobile ad-hoc routing, and propose a generic architecture and API for implementing ad-hoc routing protocols in modern OS. We implement these services and API in Linux. With the helps of standard Linux primitives, we were able to do so without any modification in the core kernel source, i.e., without the need to recompile the kernel. The software consists of a user-space library (ASL) and protocol dependent loadable kernel modules. To demonstrate the flexibility of this approach, we implement AODV using ASL. We also give detailed design for implementing other routing protocols in this framework, and share our experience in implementing these different protocols.

We believe that the principle of separating routing and forwarding has profound importance in ad-hoc routing system design. Protocols that follow this principle are easier to implement in a clean way. The code will be efficient, portable, maintainable, and extensible. Protocols that violate this principle and mix routing and forwarding will require significantly more efforts to produce a clean and well-structured implementation. Unfortunately, little attention is paid in today's ad-hoc network research on the considerations of system issues in protocol design. Even if the protocol architecture follows the principle of separation, many protocols still suggest subtle optimizations that violate this rule. These optimizations are often easy to simulate but very difficult to implement in real systems. In some cases the complications encountered can nullify the benefits of the intended optimizations.

Implementing a routing protocol is very important to validate its design. Coming up with a clean implementation not only helps better understanding of the protocol nuances, but also allows extensions to explore the protocol design space. For example, many ad-hoc routing research efforts have extended upon AODV and DSR; having clean and extensible implementations of these two protocols would benefit these entire research directions.

Note: Source code for the Ad-hoc Support Library and AODV-UIUC is available under the GNU Public License from `http://aslib.sourceforge.net`. Source code for DSR implementation will also be available at this URL.

References

[1] AODV homepage. http://moment.cs.ucsb.edu/AODV/aodv.html.

[2] AODV-uppasala university. http://www.docs.uu.se/ henrikl/aodv.

[3] Implementation of DSR. http://www.monarch.cs.cmu.edu/dsr-impl.html.

[4] Kernel AODV. http://w3.antd.nist.gov/wctg/aodv_kernel/.

[5] Netfilter/Iptables homepage. http://www.netfilter.org.

[6] TORA/IMEP. http://www.cshcn.umd.edu/tora.shtml.

[7] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. D. Kim, B. Zhou, and E. G. Sirer. On the need for system-level support for ad hoc and sensor networks. *ACM Operating Systems Review*, 36(2):1–5, Apr. 2002.

[8] S. Corson, S. Papademetriou, P. Papadopoulos, V. Park, and A. Qayyum. An internet MANET encapsulation protocol (IMEP) specification, Aug 1999. IETF Draft, draft-ietf-manet-imep-spec02.txt, work in progress.

[9] S. Desilva and S. Das. Experimental evaluation of a wireless ad hoc network. In *Proceedings of the 9th International Conerence. on Computer Communications and Networks*, 2000.

[10] S. Doshi, S. Bhandare, and T. X. Brown. An on-demand minimum energy routing protocol for a wireless ad hoc

network. *Mobile Computing and Communications Review*, 6(2), July 2002.

[11] B. Gupta. Design, implementation and testing of routing protocols for mobile ad-hoc networks. Master's thesis, University of Illinois at Urbana-Champaign, 2002.

[12] Z. J. Haas. The routing algorithm for the reconfigurable wireless networks. In *Proceedings of IEEE International Conference on Universal Personal Communications (ICUPC'97)*, San Diego, California, Oct. 1997.

[13] Y.-C. Hu and D. B. Johnson. Implicit source routes for on-demand ad hoc network routing. In *Proceedings of the 2001 ACM International Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc'01)*, pages 1–10, Long Beach, California, Oct. 2001.

[14] D. Johnson, D. Maltz, Y.-C. Hu, and J. Jetcheva. The dynamic source routing protocol for mobile ad hoc networks (DSR). IETF Internet-Draft, draft-ietf-manet-dsr-07.txt, Feb. 2002.

[15] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In T. Imielinski and H. Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[16] F. Kargl, J. Nagler, and S. Schlott. Building a framework for manet routing protocols. URL: http://medien.informatik.uni-ulm.de/~frank/research/manetframework.pdf.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[18] U. C. Kozat, G. Kondylis, B. Ryu, and M. K. Marina. Virtual dynamic backbone for mobile ad hoc networks. In *Proceedings of IEEE ICC'01*, 2001.

[19] D. Maltz, J. Broch, and D. Johnson. Experiences designing and building a multi-hop wireless ad hoc network testbed. Mar. 1999.

[20] D. A. Maltz. *On-Demand Routing in Multi-hop Wireless Mobile Ad Hoc Networks.* PhD thesis, Carnegie Mellon University, 2001.

[21] D. A. Maltz, J. Broch, J. Jetcheva, and D. B. Johnson. The effects of on-demand behavior in routing protocols for multi-hop wireless ad hoc networks. *IEEE Journal on Selected Areas in Communications*, 17(8):1439–1453, August 1999.

[22] N. K. Palanisam. Modular implementation of temporally ordered routing algorithm. Master's thesis, University of Colorado, 2001.

[23] V. Park and S. Carson. Temporally-ordered routing algorithm (TORA) version 1 functional specification. IETF Internet-Draft, draft-ietf-manet-tora-04.txt, July 2001.

[24] V. D. Park and M. S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *Proceedings of IEEE INFOCOM*, 1997.

[25] C. Perkins and E. Royer. Ad hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, New Orleans, LA, Feb. 1999.

[26] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *Proceedings of ACM SIGCOMM'94*, London, U.K., Sept. 1994.

[27] C. E. Perkins, E. M. Royer, and S. R. Das. Ad hoc on demand distance vector (AODV) routing. IETF Internet-Draft, draft-ietf-manet-aodv-10.txt, work in progress, Jan. 2002.

[28] L. L. Peterson and B. S. Davie. *Computer Networks.* Morgan Kaufmann Publishers, 2nd edition, 2000.

[29] E. M. Royer and C. E. Perkins. An implemenatation study of the aodv routing protocol. In *Proceedings of the IEEE Wireless Communications and Networking Conference*, 2000.

[30] C.-K. Toh. A novel distributed routing protocol to support ad hoc mobile computing. In *Proceedings of IEEE 15th Annual International Conference on Computers and Communications*, pages 480–486, Phoenix, March 1996.

[31] C.-K. Toh. Long-lived ad hoc routing based on the concept of associativity. IETF Internet-Draft, draft-ietf-manet-longlived-adhoc-routing-00.txt, Mar. 1999.

[32] A. Tornquis. A modular framework for implementing ad hoc routing protocols. Master's thesis, University of Colorado, 2000. http://systems.cs.colorado.edu/Networking/modular-adhoc.html.

Predictive Resource Management for Wearable Computing

Dushyanth Narayanan[†] and M. Satyanarayanan[†‡]
[†]*Carnegie Mellon University* and [‡]*Intel Research Pittsburgh*
{bumba,satya}@cs.cmu.edu

Abstract

Achieving crisp interactive response in resource-intensive applications such as augmented reality, language translation, and speech recognition is a major challenge on resource-poor wearable hardware. In this paper we describe a solution based on *multi-fidelity computation* supported by *predictive resource management*. We show that such an approach can substantially reduce both the mean and the variance of response time. On a benchmark representative of augmented reality, we demonstrate a 60% reduction in mean latency and a 30% reduction in the coefficient of variation. We also show that a *history-based* approach to demand prediction is the key to this performance improvement: by applying simple machine learning techniques to logs of measured resource demand, we are able to accurately model resource demand as a function of fidelity.

1 Introduction

Resource-intensive applications such as speech recognition, language translation, and augmented reality pose a dilemma for wearable computing. Such applications are valuable because they support hands-free interaction. However, their peak resource demands can overwhelm the processing speed, memory, and battery capacity of wearable hardware whose weight, size and form factor are limited by user comfort. The result is sluggish interactive response that can seriously distract a mobile user engaged in a physically and cognitively demanding task such as bridge inspection, aircraft maintenance or military action.

Technology improvements through Moore's Law will not solve this problem. Rather, it is likely to persist because market forces in wearable computing demand continuous improvements in user comfort rather than just improvements in compute power. This tension leads to the question addressed by this paper: *How can we achieve crisp interactive response for resource-intensive applications on wearable computers?*

In this paper, we show how *multi-fidelity computation* can help to bound interactive latency by dynamically trading resource demand for output quality, or *fidelity*. We describe the design, implementation and evaluation of a system that supports multi-fidelity computation. The system automatically makes runtime fidelity decisions on the applications' behalf, thus freeing programmers from this burden. To make sound fidelity decisions, it exploits *history-based prediction* of application resource usage.

Our implementation is based on Odyssey [15, 30], which originally supported the concept of fidelity for stored data. This work extends that concept to the broader notion of computational fidelity and demonstrates its applicability to a new class of applications. In the rest of this paper, the term "fidelity" will mean "computational fidelity" and "Odyssey" will refer to the multi-fidelity support added by us to the base system.

We have experimentally validated our approach using four applications. Because of space limitations, we only describe one application case study in detail here, and summarize the results of the other three. Full details of the latter can be found in Narayanan's dissertation [27]. Our key results can be summarized as follows:

- Predictive resource management can bound response latency and reduce its variability.
- History-based prediction of resource demand is feasible, accurate, and necessary for this improvement.
- Legacy applications can be ported at modest cost to a multi-fidelity programming model.

Section 2 describes our high-level design principles and rationale. Section 3 describes our prototype API for multi-fidelity computation, and the implementation of the runtime support layer. It also explains our methodology for constructing application-specific resource demand predictors, and de-

This research was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Materiel Command (AFMC) under contracts F19628-93-C-0193 and F19628-96-C-0061, the Space and Naval Warfare Systems Center (SPAWAR) / U.S. Navy (USN) under contract N660019928918, the National Science Foundation (NSF) under grants CCR-9901696 and ANI-0081396, IBM Corporation, Intel Corporation, Compaq, and Nokia. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies or endorsements, either express or implied, of DARPA, AFMC, SPAWAR, USN, the NSF, IBM, Intel, Compaq, Nokia, Carnegie Mellon University, the U.S. Government, or any other entity.

[†] Dushyanth Narayanan is currently a researcher at Microsoft Research Ltd., Cambridge, UK and can also be reached at `dnarayan@microsoft.com`.

scribes one example in detail. Section 4 presents a comprehensive evaluation of the system: we measure the accuracy of history-based prediction, the performance benefits of predictive resource management, and the programming costs and runtime overheads. Section 5 describes related work, and Section 6 concludes with some directions for future research.

2 Design rationale

2.1 Alternatives

There are three fundamentally different approaches to coping with situations where application resource demand exceeds supply. One approach is to prevent such situations by using *QoS-based resource reservations* [22, 26]. For example, an application may be able to reserve a minimum fraction of a CPU and thus guard against insufficient supply of this resource due to competition from concurrent applications. As another example, it may be possible to reserve bandwidth in a carefully controlled networking environment. Unfortunately, enforcement of QoS-based reservations requires operating system support that is rarely present in standard OS distributions. More importantly, this approach fails when the peak resource demand of a single application exceeds the capabilities of the hardware it is running on.

The second approach is to acquire additional resources through *remote execution.* Even a resource-impoverished wearable computer such as the IBM Linux wristwatch [29] can use compute servers to run resource-intensive applications. In previous work, we described Spectra [14], a remote execution subsystem layered on the multi-fidelity framework described here. We are further exploring remote execution in current work [4]. However, there are many situations in which a mobile user has no access to compute servers and must therefore rely solely on the resources of his wearable computer. A different approach must be used to handle those situations.

The third approach is to reduce resource demand through multi-fidelity computation. As its name implies, multi-fidelity computation assumes that an application is capable of presenting results at different fidelities. Users prefer results of higher fidelity, but can tolerate results of lower fidelity. A high-fidelity result requires greater resources to compute than a low-fidelity result. When resources are plentiful, the application generates high-fidelity results; when resources are scarce, it generates low-fidelity results. By dynamically varying fidelity, timely results can be generated over a wide range of resource levels. We elaborate on this in the next section.

2.2 Multi-fidelity computation

The classic notion of an algorithm has a fixed output specification but variable resource demand. In contrast, it is the output specification that is variable in a multi-fidelity computation [32]. By setting runtime parameters called *fidelity metrics*, we can obtain different outputs for the same input. One can say, in effect, "Give me the best result you can using no more than X units of resource R." R is typically response latency in an interactive application, but it can also refer to memory, energy, bandwidth or any other resource. Thus multi-fidelity computations are a generalization of any-dimension algorithms [25]. The latter can be viewed as multi-fidelity computations which incrementally refine their output, allowing them to be interrupted at any point to yield a result.

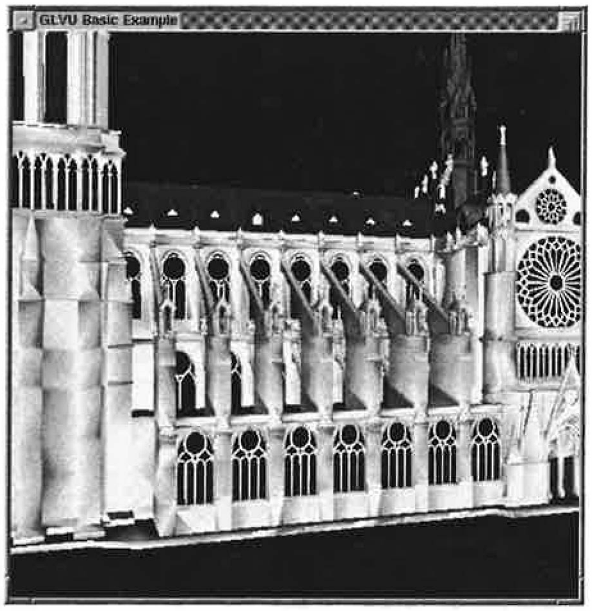
Multi-fidelity computation allows us to choose the best runtime tradeoff between output quality and performance. In an interactive application, each interactive operation can be viewed as a multi-fidelity computation. At the beginning of each operation, its fidelity metrics can be set to yield the desired response latency at the current resource availability.

2.3 Motivating example

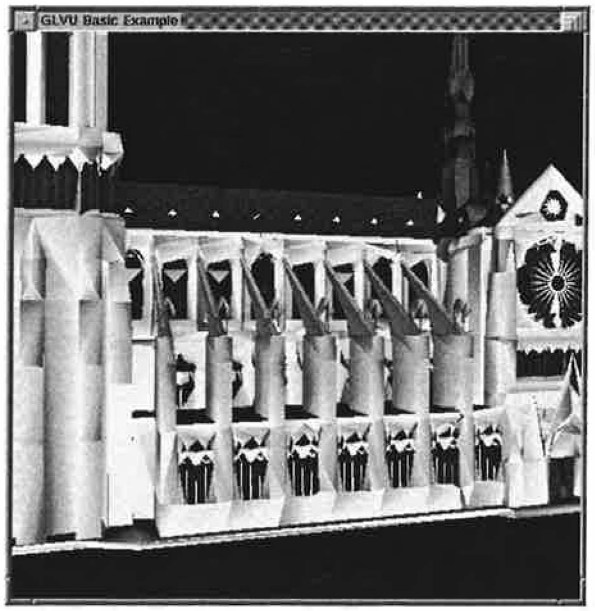
Throughout this paper we will use *augmented reality (AR)* [3] as the driving example to illustrate various aspects of our system. Although AR is a relatively young technology, it has already proved useful in a number of domains such as tourist guides [12], power plant maintenance [11], architectural design [37], and computer-supported collaboration [5].

In AR, a user looks through a transparent heads-up display connected to a wearable computer. Any displayed image appears to be superimposed on the real-world scene before the user. AR thus creates the illusion that the real world is visually merged with a virtual world. This requires a precise correspondence between the two worlds. As a user's orientation and location change, the displayed image must rapidly and accurately track those changes. Sluggish tracking can be distracting to the user and, in extreme cases, can result in symptoms similar to sea-sickness.

3-D rendering, a computationally intensive operation, lies at the heart of AR. Even a brief turn of the head by a user can result in a complex scene having to be re-rendered multiple times. For example, an architect might use AR for on-site design. This would allow her to visualize the impact of proposed design changes such as new windows or color schemes. Before converging on a final design, she may iteratively try out many alternatives, viewing them from different angles and under different hypothetical lighting conditions such as moonlight or sunset.



High fidelity (1.0)



Low fidelity (0.1)

Figure 1: Effect of fidelity on 3-D rendering

3-D rendering for AR embodies many of the characteristics that motivate the work described in this paper. First, it is extremely resource intensive, particularly of CPU and memory. Second, to be fully effective it must run on a lightweight wearable computer. Third, crisp interactive response is critical. Fourth, there is a fidelity metric, the resolution of the displayed image, that directly impacts resource consumption.

Figure 1 illustrates the last point. The high-fidelity figure on the left contains ten times as many polygons as the low-fidelity figure on the right. Since CPU demand increases with the number of polygons, the low-fidelity figure can be rendered much faster. In many situations, the low-fidelity figure may be acceptable; the user can always explicitly ask for re-rendering at higher fidelity.

2.4 Predictive resource management

Before executing an interactive operation, an application must determine its fidelity settings. Odyssey serves as an oracle in making this decision. Its recommendation is based on a search of the space of fidelity settings. This search requires Odyssey to predict resource supply during the operation, as well as resource demand and operation latency for different settings. It also requires Odyssey to correctly reflect the user's current preferences in the tradeoff between output quality and operation latency.

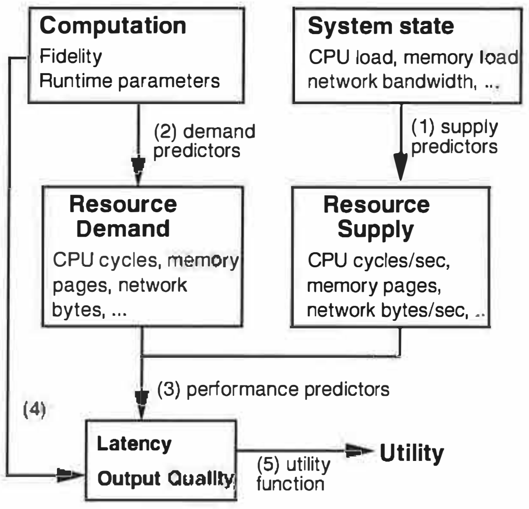


Figure 2: Mapping fidelity to utility

As Figure 2 shows, the complete prediction process can be decomposed into five predictive mappings. Three of the mappings relate to predicting operation latency: (1) from system load statistics to resource supply; (2) from fidelity to resource demand; and, (3) from resource supply and demand to operation latency. The other two mappings translate fidelity and latency predictions into predictions of user satisfaction or *utility:* (4) from fidelity to output quality; and, (5) from latency and output quality to utility.

Odyssey performs mapping 1 using *supply predictors* that monitor kernel load statistics through standard interfaces, and make inferences based on gray-box knowledge [2] of kernel resource management policies. It uses history-based demand predictors to perform mapping 2, and performance predictors based on a resource model for mapping 3. These components are described in Sections 3.3 and 3.4.

Mapping 4 specifies the output quality we can expect at each setting of each fidelity "knob". In general, this mapping would be determined through studies of user perception. In this work, we make the simplifying assumption that fidelity and output quality are synonymous; in other words, that the mapping is trivial. For example, we use the "JPEG level" parameter of a JPEG compression algorithm as a measure of the output image quality; the work by Chandra and Ellis [6] confirms that this is acceptable. Mapping (5) is a *utility function* that captures current user preferences. In a deployed system, utility functions would be generated automatically from a GUI or by inferring user intent. In our experimental prototype, we use the hand-crafted utility functions described in Section 3.5.

3 Interface and implementation

3.1 Programming interface

The multi-fidelity programming model is based on the notion of an *operation.* An operation is the smallest user-visible unit of execution, from user request to system response. Each operation corresponds to one multi-fidelity computation, with fidelity metrics settable at operation start. Examples include rendering an augmented reality scene; recognizing a speech utterance and displaying the recognized text; fetching and displaying a web image.

Figure 3 shows the basic multi-fidelity API. *register_fidelity* is called at application startup. Odyssey then reads an Application Configuration File (ACF), which specifies the multi-fidelity operation type, its fidelity metrics, and their value ranges (Figure 4).

```
int register_fidelity(IN char *conf_file,
     OUT int *optype_idp);

int begin_fidelity_op(IN const char *dataname,
     IN int optype_id,
     IN int num_params,
     IN fid_param_val_t *params,
     IN int num_fidelities,
     OUT fid_param_val_t *fidelities,
     OUT int *opidp);

int end_fidelity_op(IN int optype_id,
     IN int opid,
     IN failure_code failed);
```

C function prototypes for the API described in Section 3.1.

Figure 3: The Odyssey multi-fidelity API

```
description glvu:render
logfile /usr/odyssey/etc/glvu.render.log
constraint latency 1.0
param polygons ordered 0-infinity
fidelity resolution ordered 0.01-1

hintfile /usr/odyssey/lib/glvu_hints.so
hint cpu glvu_render_cpu_hint
update glvu_render_update
utility glvu_render_utility
```

The first five lines specify a descriptive tag for the operation; a pathname for writing log data; a target latency for the operation; a non-tunable parameter; and a fidelity metric. The last four lines specify a binary hint module and name its entry points.

Figure 4: Application Configuration File for rendering

The ACF also specifies *nontunable parameters*: runtime variables such as input data size that affect resource demand, but are not adaptable. For example, the resource demand of rendering depends not only on the resolution, but also on the polygon count of the original, full-resolution scene.

Finally, the ACF specifies an application-specific *hint module*. This binary module contains the application-specific resource demand predictors and the user utility function. For efficient runtime invocation of the demand predictors and utility function, the hint module is loaded into Odyssey's address space. We are looking at ways to retain the efficiency, but improve on the safety, of this approach.

Before each operation, the application invokes *begin_fidelity_op*, and passes in the nontunable parameters. Odyssey computes and returns the optimal fidelity value(s) for the operation. After each operation, the application calls *end_fidelity_op*. Odyssey then logs the operation's measured resource demand: these logs are used for history-based resource demand prediction (Section 3.4).

3.2 System architecture

Odyssey is implemented as a user-level process on a standard Linux 2.4 kernel. Its primary functionality — making fidelity decisions — is triggered by *begin_fidelity_op* and implemented in the following steps, numbered as in Figure 5:

1. The application passes in the nontunable parameters.
2. *Supply predictors* estimate the application's resource supply for the near future (mapping 1 of Figure 2).
3. An iterative *solver* searches the fidelity space for the best candidate.
4. *Demand predictors* map fidelity to resource demand (mapping 2).
5. A *performance predictor* estimates latency given supply and demand predictions (mapping 3).
6. A *utility function* evaluates the proposed fidelity-performance tradeoff (mapping 5).
7. After several iterations of steps 3–6, the solver returns the fidelity with the highest utility.

The system's second function — monitoring and logging — is triggered by *end_fidelity_op*:

8. *Demand monitors* measure the resources consumed by the just-concluded operation.
9. A *logger* records the resource demand, fidelity, and nontunable parameter values to a disk file.
10. These values are also passed to the demand predictors, to update their predictive models.

Section 3.3 describes the generic system components: the supply predictors, performance predictors, solver, demand monitors, and logger. Demand predictors are application-specific: Section 3.4 describes our history-based method for constructing them. Section 3.5 then describes our approach to constructing utility functions.

3.3 Generic system components

3.3.1 Supply predictors

Our prototype has supply predictors for CPU, memory, network, energy and file cache. Each of these monitors kernel statistics, and makes predictions of resource availability for each application at the beginning of each operation. For brevity, we only describe the CPU supply predictor here.

The CPU supply predictor predicts, at the start of each operation, the CPU supply available to it in cycles/sec. It is based on some simplifying assumptions: that the operation is single-threaded; that all CPU-bound processes receive equal shares; that I/O-bound processes offer negligible CPU load; and that past load predicts future load at all time scales. These assumptions give us a simple predictor: a process p's CPU supply over the next T seconds is

$$S_{cpu} = \frac{P}{N+1}$$

where P is the processor clock speed, and N is the predicted background load over the next T seconds: that is, the average number of runnable processes other than p. We periodically sample the instantaneous load average n_i from **/proc/loadavg**, and subtract out p's contribution, $n_i(p)$. The latter is 1 if p is runnable, and 0 if not. We then smooth the samples:

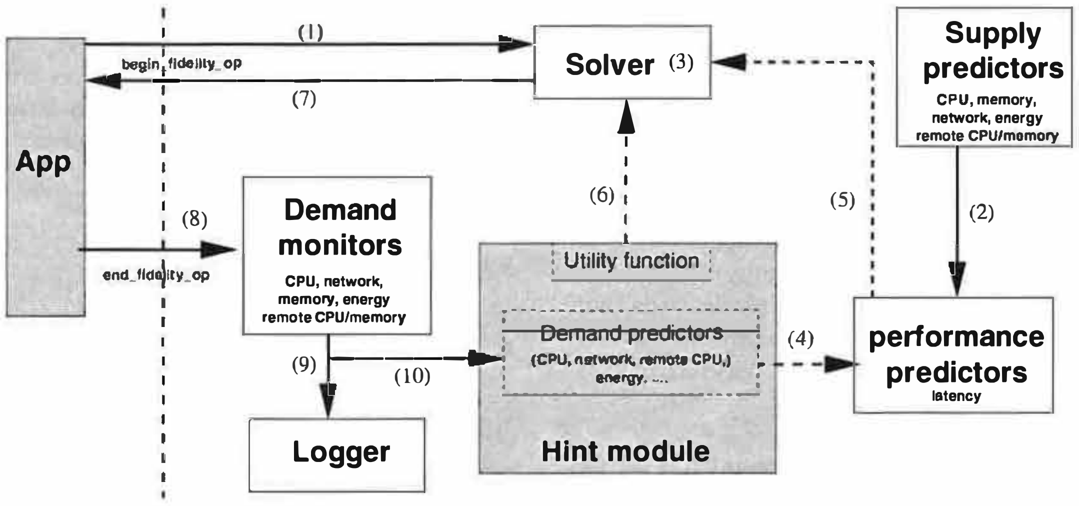
$$N_{i+1} = \alpha N_i + (1-\alpha)(n_i - n_i(p))$$

We set

$$\alpha = e^{\frac{-t_p}{T}}$$

where t_p is the load sampling period, 0.5 s in our prototype. This makes the decay time equal to the prediction horizon T. In other words, we use more history for predictions over longer periods.

The clock speed P is read from **/proc/cpuinfo** at startup. Currently Odyssey runs on a stock Linux kernel without dynamic clock scaling support. When such support is available, it should be possible to update P dynamically from **/proc** whenever the clock speed changes.



Shaded boxes represent application-specific components; components to the right of the dotted line are part of Odyssey. The arrows show the data flow between components; dashed arrows correspond to interactions that occur many times for a single invocation of the API. The numbers correspond to the steps in Section 3.2.

Figure 5: System support for the multi-fidelity API

3.3.2 Performance predictors

Our current prototype has predictors for two performance metrics: operation latency and battery drain [15]. Here we focus on operation latency, the key metric for interactive applications.

Our latency predictor computes latency as a function of resource supply and demand. It is based on a simple resource model that assumes *sequential* use of resources (no overlapping of processing and network I/O). It computes latency as:

$$L = \frac{D_{local\ cpu}}{S_{local\ cpu}} + \frac{D_{xmit}}{S_{xmit}} + \frac{D_{recv}}{S_{recv}} + \frac{D_{rtt}}{S_{rtt}} + \frac{D_{remote\ cpu}}{S_{remote\ cpu}}$$

Here $S_{local\ cpu}$ is the predicted CPU supply in cycles/sec available to the application. $D_{local\ cpu}$ is the predicted CPU demand in cycles required by the operation. The other terms represent the time taken for a remote execution(s): transmitting data to a server, receiving results from it, round trip time of one or more RPCs, and server-side computation. Network bandwidth and round trip time estimates are provided by the base Odyssey infrastructure [30]. The predictor also computes the effects of VM paging and remote file access [27]; for brevity, we do not discuss these.

The default generic latency predictor can be overridden at runtime by an application-specific predictor: for example, one that allows for overlapping computation and I/O.

3.3.3 Solver, demand monitors, and logger

The solver searches the space of fidelities and finds the values that maximize utility. It uses a gradient-descent strategy for numeric parameters, and exhaustive search for non-numeric parameters such as enumerated lists. It works well for applications with a small number of fidelity metrics and well-behaved utility functions without multiple local maxima; we could easily substitute more robust and scalable algorithms such as Lee's [22] without modifying other system components.

Demand monitors measure the resource demand of each operation based on kernel statistics from **/proc**. For example, CPU demand is the CPU time used by an operation, scaled by the processor clock speed. This information is written to a disk file by the logger.

3.4 History-based demand predictors

A key component of our architecture is the demand predictor: a function that maps an operation's fidelities and nontunable parameters to its resource demand, in units independent of runtime system state such as load or clock speed. For example, CPU demand is measured in cycles consumed per operation.

We construct demand predictors empirically from application history logs [28], rather than relying exclusively on static analysis. First, the application programmer or domain expert

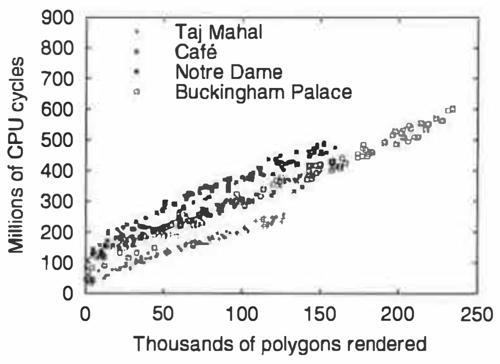


Figure 6: CPU demand of rendering (fixed camera, varying resolution)

identifies fidelity metrics and other runtime parameters affecting resource demand. From a static analysis, they might also give a functional form relating these parameters to resource demand: for example, "CPU demand is quadratic in input data size".

The remaining steps are automated, requiring little or no user intervention: we run the computation at different parameter values, and Odyssey automatically logs each operation's resource demand. We use statistical machine learning techniques to fit the logged data to the functional form, generating a predictive mapping function. At runtime, we continue to refine this function using online learning techniques.

Although demand predictors are application-specific, we believe our methodology will allow their construction by third parties without extensive domain expertise. Additionally, demand predictors are separate code modules, and do not require modification of the application source code. We illustrate our method through one detailed example, and describe two techniques that proved extremely useful in improving predictor accuracy. Section 4.3 evaluates prediction accuracy for our chosen example as well as for other applications and resources.

3.4.1 Example: CPU demand predictor for rendering

Rendering is CPU-bound, and good interactive response depends on accurate prediction and regulation of CPU demand. For our rendering algorithm, *resolution* is the fidelity metric: thus we need to know the mapping from resolution to CPU demand. CPU demand depends both on the resolution r and the original polygon count p; from examining the algorithm, we expected in fact that it would be a function of pr, the rendered polygon count.

To map resolution to CPU demand, we started by logging the CPU demand at different resolutions for four different scenes, and plotting CPU demand against rendered polygon count (Figure 6). We see that CPU demand is *linear* in rendered polygon count:

$$D_{cpu} = c_0 + c_1 pr$$

for a fixed scene and camera position (note that different scenes have different values of c_0 and c_1). However, the scene and the camera position are parameters that can vary at runtime, and must be tracked. In the following sections, we show how we track this variation using *data-specific prediction* and *online learning*.

3.4.2 Data-specific prediction

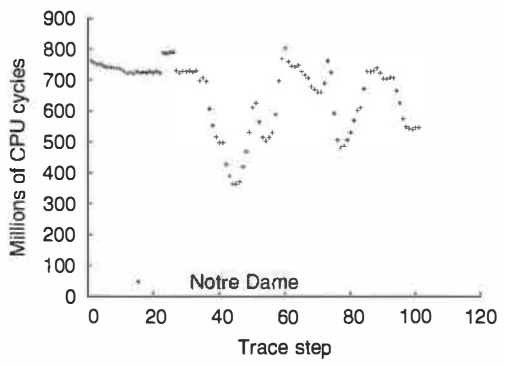
Sometimes resource demand depends on data-specific effects other than the data size, which are not easily expressed as numeric parameters. For example, the CPU demand of rendering depends on the contents of the scene being rendered. In such cases, data-specific prediction can be extremely useful: maintaining separate predictor coefficients for each data objects. Sometimes, these can be computed offline and stored with the data: for example, JPEG [36] compression ratios depend on image content, and these "compressibility coefficients" could be precomputed and stored at the web server.

In other cases, the data-specific coefficients must be computed online, after observing the resource demand of a few operations on a new data object. This can still be useful if we perform many operations on the same object: for example, with rendering, the user will usually navigate a single scene for a while.

3.4.3 Online learning

Sometimes, we may have portions of application state which affect resource demand but are not easily used as part of a predictive model. For example, the CPU demand of rendering depends not only on the resolution and the scene, but also on the camera position. Figure 7 shows that the CPU demand of rendering varies considerably with camera position as a user navigates a scene, even when fidelity is fixed.

Thus, camera position and orientation are nontunable parameters affecting CPU demand. Unfortunately, their effect on CPU demand is very complex, depending on local properties



The graph shows the CPU demand of rendering the Notre Dame scene over time, at a resolution of 1. Each point corresponds to one camera position in a motion trace of a user navigating the scene. All experiments were run on the hardware described in Section 4.1.

Figure 7: CPU demand of rendering (fixed resolution, moving camera)

of the scene: mapping them directly to CPU demand requires large and expensive lookup tables. Instead, we use a much simpler technique based on the observation that

- At each camera position, the linear relationship $D_{cpu} = c_0 + c_1 pr$ holds, but c_0 and c_1 vary with camera position.
- In typical use, camera position changes incrementally: the user follows a continuous path through the scene.
- CPU demand has locality: a small change in camera position results in a small change to c_0 and c_1.

We use an *online-learning* method that uses the linear mapping $D_{cpu} = c_0 + c_1 pr$, but continuously updates the values of c_0 and c_1 to reflect the behaviour corresponding to the current camera position. We use recursive least-squares regression with exponential decay [39], a modification of the well-known linear regression method [18]. This gives greater weight to more recent data by decaying the weight of data exponentially over time. Our predictor uses a decay factor of 0.5, which makes it very agile, effectively remembering only the last 4 data points. It is also cheap: a 2-dimensional linear fit requires only tens of bytes of state, and tens of floating point instructions per update.

The online-learning predictor is also data-specific. For each new scene, it initializes a predictor with generic coefficients computed from a variety of scenes and camera positions. Subsequent renders of that scene result in updates of the scene-specific predictor, specializing it both for the scene and the camera position within the scene. In Section 4.3 we show that these two simple techniques improve prediction accuracy significantly for rendering; we believe that they have more general applicability as well.

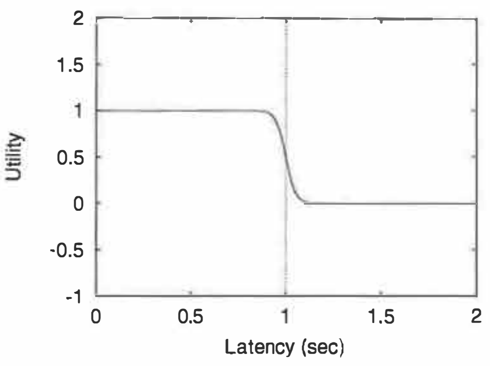


Figure 8: Sigmoid utility function

3.5 Utility functions

Utility functions represent a user's tradeoff policy between fidelity and performance. Given some estimated fidelity and performance, the utility function returns a number in $[0, 1]$ representing the resulting user happiness; 0 represents the least possible user satisfaction and 1 the most. By default, we use linear functions for utility as a function of fidelity, and *sigmoids* for utility as a function of latency. The product of these functions gives us a multidimensional utility function whose range is still $[0, 1]$. In Odyssey, utility functions are computed by binary code modules; the user can override the default utility function with an arbitrarily general one by providing their own module.

A sigmoid is a smoothed version of a step function. Instead of having utility fall off a cliff when latency exceeds its target value, we can now specify a tolerance zone where latency degrades linearly. Figure 8 shows a sigmoid with a target of 1 s and a tolerance of 10%. There is little gain in utility from decreasing latency below 0.9 s: this is the *sweet spot* of the curve. Above 0.9 s, utility decreases steadily, and latencies above 1.1 s are unacceptable to the user.

4 Evaluation

This section validates the predictive resource management approach by answering three sets of questions:

- Is history-based demand prediction accurate? Are data-specific prediction and online learning useful?
- How does predictive resource management improve performance? Can multiple concurrent applications adapt successfully without interfering with each other?
- What are the programming costs and runtime overhead of using the system?

Before we describe the experiments that answer these questions, we first describe our experimental setup (Section 4.1) and evaluation metrics (Section 4.2). Sections 4.3–4.5 then address each of the above sets of questions in turn.

4.1 Experimental platform and benchmarks

Our platform for all experiments reported in this paper is an IBM ThinkPad 560 with a 233 MHz Mobile Pentium MMX processor, 96 MB of RAM, no 3-D graphics hardware, and running a standard Linux 2.4.2 kernel. We used this rather than a wearable computer for ease of development and testing; its processing power is comparable with recent wearable and handheld platforms such as the IBM Linux watch [29] and the Compaq iPAQ 3650.

Our motivating example — augmented reality — is not a mature technology, and fully fledged AR applications are not freely available. Instead, we use as benchmarks two applications — GLVU and Radiator — which provide one component of AR: 3-D rendering. Together, these applications approximate the augmented reality scenario of Section 2.3: an architect using AR for on-site design.

GLVU [35] is a "virtual walkthrough" program that allows a user to explore a virtual 3-D scene: its function is to render the scene from any viewpoint chosen by the user. In our experiments, we simulate a moving user by replaying a trace of a user navigating a 3-D scene using GLVU's graphical user interface. We assume a continually moving user, and do not insert any think times between render requests.

Radiator [38] computes lighting effects for 3-D rendering using radiosity algorithms [8]. In an AR scenario, it would be re-run whenever the user modified the scene lighting, for example by adding a window to a building. We simulate this user behaviour by running sporadic radiosity computations during the virtual walkthrough, with random intervening think times.

Both GLVU and Radiator support *multiresolution scaling* [17], which allows each render or radiosity computation to be done at any resolution — any fraction of the original polygon count. The overhead of changing the resolution is negligible. Resolution is thus the single fidelity metric for both computations.

In a real AR application, the user would be able to interactively edit the scene, and the lighting effects computed by Radiator would be fed back into GLVU for rendering. In our version, GLVU and Radiator lack interactive editing facilities and do not communicate with each other. However, the benchmarks are representative of AR from a resource and performance point of view.

4.2 Evaluation metrics

Demand predictor *accuracy* is measured by running an application benchmark on an unloaded system, and measuring the relative error for each operation: the difference between the predicted and observed resource demand, divided by the latter. We use relative rather than absolute prediction error since it is applicable across a wide range of values. Given the relative error for a number of operations, we report the *90th percentile* error E_{90}. An E_{90} of 5 % means that 90% of the time, the predictor was within 5 % of the correct value.

Our metric of interactive application performance is operation latency. Specifically, we measure Odyssey's ability to keep latency within user-specified bounds, with low variability and without unnecessarily sacrificing fidelity. In other words, we measure the ability of the adaptive mechanism — Odyssey — to implement one kind of policy: keeping latency steady. Our adaptive policies are implemented by a sigmoidal utility function centred on the desired latency bound (Section 3.5), with a tolerance of 10%. Utility also increases linearly with fidelity. The net effect is that utility is maximized at 90% of the latency bound: this is the *target* latency.

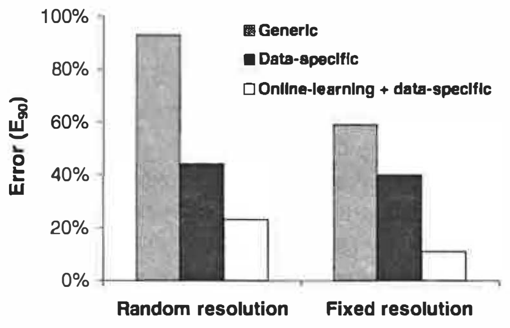
We conduct 5 trials of each experimental run. For each such set of 5 trials, we report the mean operation latency, and also the coefficient of variation: the standard deviation of latency divided by the mean. In some cases, we also show a timeline of one of the trials, to illustrate the performance and fidelity over time.

Ideally, we want mean latency to be on target. Higher latencies indicate bad interactive response, while lower latencies indicate an unnecessary sacrifice of fidelity. We also want the coefficient of variation to be small: variability in performance leads to a bad user experience [24]. High variation also indicates that the system is often off-target: in other words, not implementing the adaptive policy well.

4.3 Demand predictor accuracy

In this section, we show that history-based demand predictors provide accurate predictions across a range of applications and resources. For brevity, we describe in detail only the CPU demand predictor for GLVU, and summarize results for other predictors.

For GLVU, we measured the accuracy of the data-specific, online-learning predictor, and also the contribution of data-specificity and online learning to this accuracy. We compared



We show 90th percentile error (in %) of three different schemes for predicting the CPU demand of rendering. The CPU demand itself varies between 0.1 s and 5 s (23–1018 million cycles).

Figure 9: CPU demand prediction error for rendering

- a *generic* predictor, which fits a single pair of coefficients c_0, c_1 to all 4 scenes,
- a *data-specific* predictor, which specializes c_0 and c_1 to each scene,
- the *online-learning* predictor, which maintains scene-specific coefficients, and also updates them after each operation to track runtime variation in CPU demand.

The accuracy of CPU demand prediction depends not only on the variation in camera position, but also on the variation in fidelity from one rendering operation to the next. In an adaptive system, variation in fidelity is driven by variation in resource supply at runtime. To estimate demand prediction accuracy independent of runtime conditions, we evaluated both the worst case of randomly varying fidelity and the best case of fixed fidelity.

Figure 9 shows the prediction error of these three predictors for both random and fixed resolution (1.0), measured on user traces on four different scenes. Each trace has 100 camera positions, yielding 400 data points in all. We see that both data-specificity and online learning decrease prediction error. The best predictor, online-learning, has a worst-case error of 24%, which is small compared to the order-of-magnitude variation in CPU demand. Better learning techniques could improve its accuracy further.

We also measured demand predictor accuracy for other applications — Radiator, speech recognition, and web browsing — and other resources — memory, network, and battery energy (Figure 10). In each case, we are able to predict to within a small fraction a quantity with a large dynamic range, showing that multi-fidelity computation can make a big difference to resource demand, and that we can predict resource demand to within a small error. Note that all the other predictors have better accuracy than the CPU predictor for GLVU: our chosen example case study represents our worst observed case.

4.4 Performance benefits

4.4.1 Single application with background load

Given that demand prediction is accurate, what is the impact on performance of predictive resource management? To answer this question, we measured the performance of GLVU adapting dynamically to changes in resource supply. GLVU plays a trace of a user navigating the "Notre Dame" scene, while Odyssey attempts to bound operation latency to 1 s. Simultaneously, a competing process alternates between spinning and sleeping every 10 s. We chose this square waveform over more realistic load patterns to explore the effect of load frequency and amplitude.

We ran this experiment in three configurations:

- *Fully adaptive*: both supply and demand prediction are enabled, so GLVU adapts to changes both in application demand and in background load.
- *Demand-only*: we enable CPU demand prediction, which allows GLVU to regulate its CPU demand to the target value. However, supply prediction is disabled: the background load is assumed to be 0.
- *Static*: GLVU's fidelity is fixed at 1: there is no adaptation at all.

Figure 11 shows one run for each configuration, in our baseline case: a trace of "Notre Dame" with a 1 s latency bound, a 0.1 Hz background load frequency, and a peak load of 1. We see that the "fully adaptive" configuration keeps latency on target. "Demand-only" is on target only when unloaded, and "static" almost never. Note that the different experiments have different run times, the effect of reducing mean latency on a fixed-work benchmark.

Figure 12 shows mean latency and variation over 5 trials for each configuration. We see that demand prediction alone substantially improves interactive performance by bringing mean latency close to the target value and reducing variability; supply prediction improves performance further.

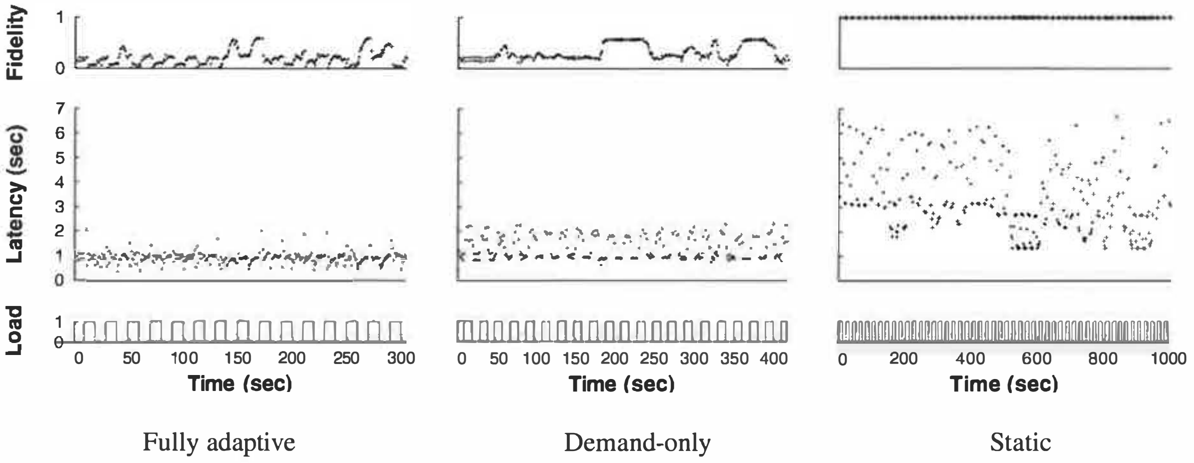
To validate our results across a range of experimental parameters, we tested the "fully adaptive" configuration with different 3-D scenes, latency bounds, and load patterns. In each case, we varied one parameter, keeping the others fixed, and compared the performance against the baseline case: Figure 13 shows the results of these experiments.

Mean latency was insensitive to experimental parameters, ex-

Application	Tunable parameters	Resource	Observed range of resource demand	Data specific?	Online learning?	E_{90}
GLVU	Resolution	CPU	23–1018 Mcycles	Yes	Yes	24%
Radiator	Resolution, algorithm	Memory	14–60 MB	No	No	3%
		CPU	220–46219 Mcycles	Yes	No	11%
Web browser	JPEG level	Energy	1.5–25 Joules	Yes	No	9%
Speech recognizer	Client-server split, vocab. size	Network	4–219 KB	No	No	0.3%
		client CPU	0–2774 Mcycles	No	No	10%
		server CPU	0–2128 Mcycles	No	No	16%

The table shows the 90th percentile error E_{90} (right-most column) of history-based demand predictors for different applications and resources. In each case, we also show the observed min-max range of resource demand, measured in millions of cycles of CPU, megabytes of memory, Joules of energy, or kilobytes of network transmission/reception.

Figure 10: Demand predictor accuracy for various applications and resources



Fidelity and latency of GLVU over time when subjected to a time-varying background load, in three different adaptation configurations. Note the different time scales on the x axes: the same benchmark takes different amounts of time in different configurations.

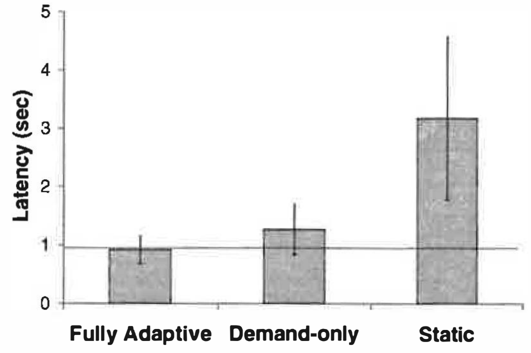
Figure 11: Adaptation in GLVU

cept when we reduced the latency bound to 0.25 s: in this case mean latency exceeds target by 20%. Here we hit the limit of fidelity degradation: on our test platform, rendering can take up to 0.46 s of CPU time even at the lowest fidelity.

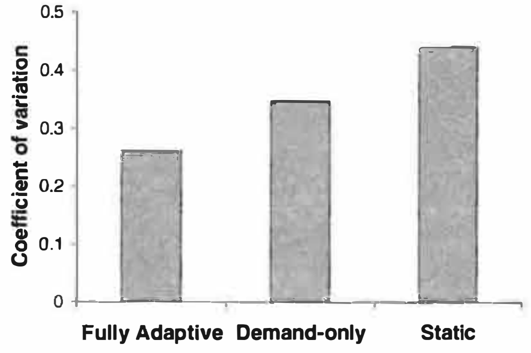
Variability in latency was the same for all scenes, but varied with other parameters. Variability was lowest for a 0.5 s latency bound. At lower latencies, Linux's 200 ms scheduler quantum causes variability. At higher latencies, load transitions are more frequent with respect to operation rate, causing more operations to deviate from target.

Variability was highest when load frequency matched operation rate (1 Hz). At lower frequencies, fewer operations are hit by load transitions. At higher frequencies, load variation gets smoothed out over the course of an operation. Variability also increases sharply with increasing load amplitude (peak-to-trough difference): operations hit by load transitions are more affected by larger transitions.

We observe that it is most important to predict resource supply *at the time scale of adaptation*: higher and lower frequencies impact latency less. If this time scale is comparable to the scheduler granularity, then prediction accuracy will be low and performance variability will be high.



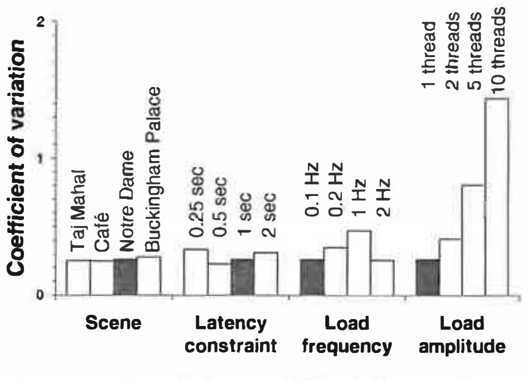
Mean latency



Variability in latency

Error bars show standard deviations; the horizontal line marks the target latency.

Figure 12: Adaptive performance in GLVU



The graph shows the coefficient of variation for latency under various experimental conditions. Each set of bars varies one parameter: the shaded bars represent the baseline case.

Figure 13: GLVU adaptation: sensitivity analysis

4.4.2 Concurrent applications

When we run two adaptive applications concurrently, are they both able to adapt effectively, or do they interfere with each other's performance? To answer this question, we mimicked an AR scenario by running GLVU and Radiator concurrently as Linux processes at default priority.

GLVU replays a trace of a user navigating the virtual "Notre Dame" scene. Meanwhile, Radiator runs sporadic radiosity computations on a copy of the same scene in the background, to simulate occasional re-computation of lighting effects by the user. Between operations, Radiator sleeps for a random "think time" from 0–10 s. The system's goal is to maintain the latency bounds of both applications despite resource variation. We use a 1 s latency bound for GLVU, as before. Radiator is much more resource-intensive, and runs in the background: for it, we use a 10 s bound. For both applications, we use a sigmoid utility function with a 10% tolerance (Section 3.5): thus, the sweet spot or target latency is 0.9 s for GLVU and 9 s for Radiator.

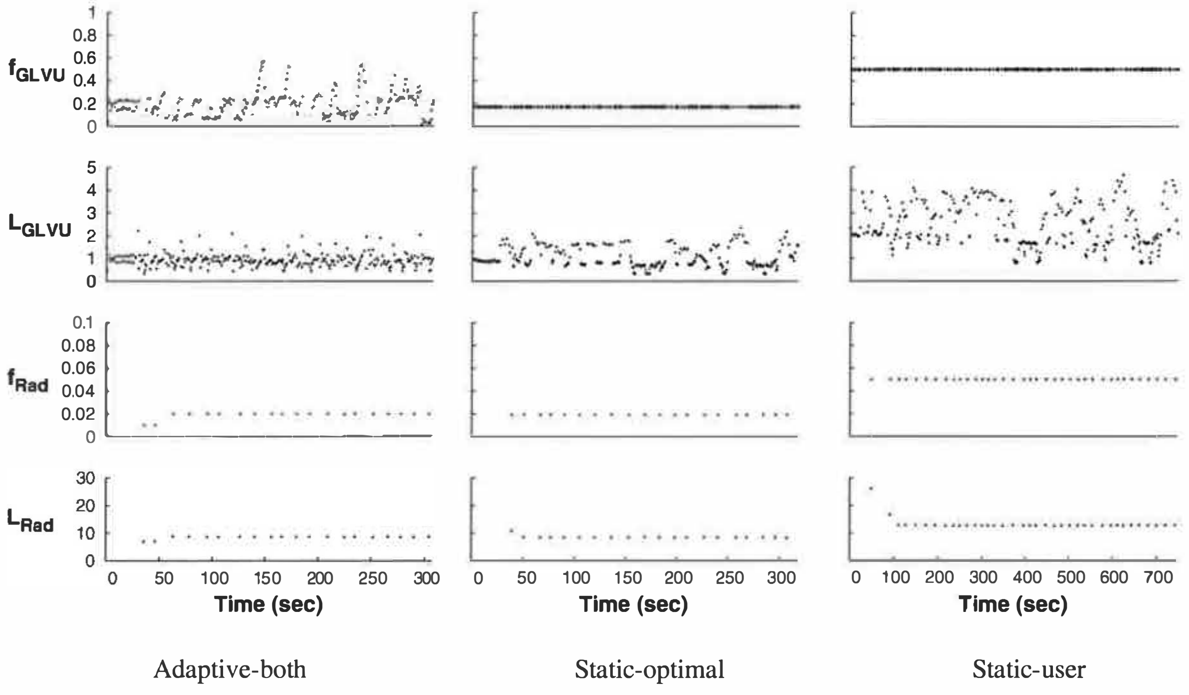
We ran this experiment in 5 configurations:

- *Adaptive-both*: both applications adapt fidelity to achieve the target latency.
- *Static-optimal*: fidelity is static, but tuned for this benchmark. We set it to the mean fidelity achieved in the adaptive case (0.17 for GLVU, 0.019 for Radiator).
- *Static-user*: fidelity is static, at 0.5 for GLVU and 0.05 for Radiator: reasonable values that a user might select without workload-specific tuning.
- *Adaptive-*GLVU: GLVU adapts, Radiator uses the "static-user" fidelity.
- *Adaptive-Radiator*: Radiator adapts, GLVU uses the "static-user" fidelity.

The last two configurations represent real-world cases where one application is modified with multi-fidelity support, but the other is unmodified and must rely on the user to set the fidelity.

Figure 14 shows one trial each for the first three configurations. In the "adaptive-both" case, GLVU maintains its target latency despite supply and demand variation. With "static-optimal", mean latency is on target but variability is high; with "static-user", mean latency is off target and variability is even higher.

For Radiator, "adaptive-both" and "static-optimal" get latency on target, while "static-user" is off target. Variability is low in all cases: Radiator's CPU demand is invariant with time and camera position. CPU supply does not vary either:



We show the performance of concurrent applications (GLVU and Radiator) over time, in three configurations. Each graph shows a time line of GLVU's fidelity (f_{GLVU}), GLVU's latency in seconds (L_{GLVU}), Radiator's fidelity (f_{Rad}), and Radiator's latency in seconds (L_{Rad}). Note the different time scale for the "static-user" graph. For lack of space, we omit the timelines for the "adaptive-GLVU" and "adaptive-Radiator" timelines: the adaptive and non-adaptive performance in these cases is very similar to that shown for the "adaptive-both" and "static-user" cases.

Figure 14: Adaptation in concurrent applications

at 10 s time scales, the competing load imposed by GLVU is constant. "Static-optimal" has slightly lower variability than "adaptive", which executes a few operations at the wrong fidelity before converging on the correct value.

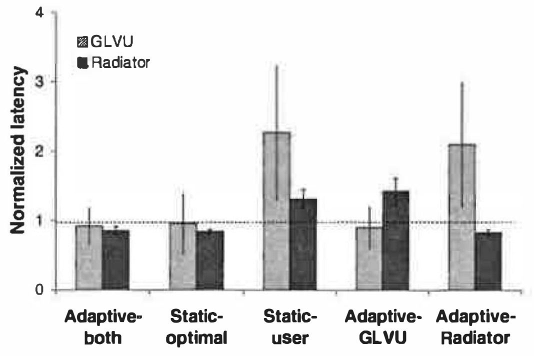
Figure 15 shows the mean normalized latency (latency divided by the latency bound) and coefficient of variation over 5 trials of all 5 configurations. We see that adaptation keeps latency on target without any workload-specific tuning, and reduces variation. Workload-specific tuning ("static-optimal") can get mean latency on target, but cannot prevent dynamic variation due to changes in resource supply or demand. Adaptation also insulates each application's performance from the other's: the "Adaptive-GLVU" and "Adaptive-Radiator" graphs show that the benefit gained from adaptation is independent of the other application's behavior. In other words, our approach can be useful even without a coordinated effort to modify all running applications. This is a valuable property for real-world deployment.

4.5 Costs and overheads

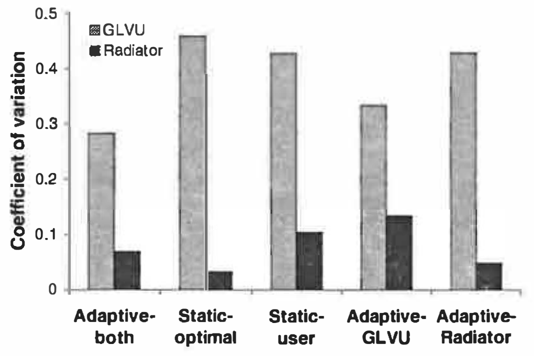
4.5.1 Porting costs

The cost of porting legacy applications to a new API is an important measure of system deployability. Figure 16 shows the amount of source code modification required for four applications to use the multi-fidelity API. Three of these already had the potential for fidelity adaptation: for example, Radiator comes with support for multiresolution models. GLVU had to be augmented with multiresolution support, and we include the cost of this step.

Multi-fidelity support requires 500–1000 new or modified lines of code, including the ACF and hint module: a modest investment of programmer effort. Many of these lines are in glue code between application constructs and the generic multi-fidelity API. We are investigating the use of stub generators to automatically generate this glue code.



Mean normalized latency



Variability in latency

Error bars show standard deviations; the horizontal line marks the target latency.

Figure 15: Adaptive performance for concurrent applications

Application	Original size		Modifications	
	KLOC[†]	Files	KLOC[†]	Files
GLVU	27.0	144	0.9[‡]	7[‡]
Radiator	51.1	222	0.6	5
Web proxy	3.9	9	0.9	6
Speech recognizer	126.4	209	1.1	10

[†] 1 KLOC = 1000 lines of code
[‡] Includes multiresolution support (0.4 KLOC, 2 files)

Figure 16: Cost of porting legacy code

Component	Overhead
App-Odyssey communication	0.36 ms
Logger (buffered at user level)	0.15 ms
Logger (unbuffered)	0.20 ms
CPU supply/demand monitor	1.38 ms
Memory supply/demand monitor	6.72 ms
Solver	10.56 ms
Total	19.37 ms

Figure 17: Per-operation runtime overhead

4.5.2 Runtime overheads

Figure 17 shows the overhead of each runtime component in additional latency per operation for a synthetic benchmark. The total overhead is around 20 ms: only 2% for a 1 s operation, but an unacceptable 20% for a 100 ms latency bound. We are looking at reducing the overheads by using a more efficient and scalable solver; with better interfaces for load and resource statistics (**/proc** contributes most of the latency to our resource predictors); and by replacing the middleware server with a library implementation.

5 Related work

This work is most closely related to previous work on fidelity adaptation [7, 9, 16, 15, 30]. We have generalized these previous notions of fidelity, which only measured data degradation, to include arbitrary runtime parameters of an application. Our system and API also move the burden of adaptation out of the application: where other systems expect applications to specify their resource requirements, we predict resource supply, demand, and performance based on observations of history. Although resource demand prediction is still application-specific, it has been cleanly separated from the rest of the system, and our history-based methodology and measurement/logging infrastructure make it an easier task than before.

We also diverge from traditional models of adaptation by using a predictive rather than a feedback-driven approach. Rather than adjusting fidelity in small steps in response to a change in performance, Odyssey can make large yet accurate adaptations in a single step. This is made possible by Odyssey's ability to predict supply, demand and performance across the entire range of fidelities.

Related, but complementary to application adaptation is work on QoS-based reservations [26, 22] and remote execution [13, 4]: Section 2.1 discussed these in more detail.

Also related is previous work on *resource prediction.* Supply prediction — predicting load from past measurements — is present in many systems. Usually it is present implicitly in a feedback loop: measurements of load or performance are used as control signals to adjust system parameters [34]. A few systems use explicit prediction of load: for example, Dinda's Running Time Advisor [10]. Explicit prediction of *resource demand*, however, is comparatively rare. Most systems assume that resource demand is constant, specified by the application, derived from a static probability distribution [23, 19], or obtained from compile-time analysis [33].

We know of two systems that explicitly predict resource demand as a function of runtime parameters: however, neither uses the predictions for application adaptation. Automated profiling for QoS [1] estimates the CPU utilization of a multimedia stream as a linear function of task rate and task size, for admission control purposes. PUNCH [21] uses machine learning to predict CPU demand as a function of application-specific runtime parameters, for load-balancing in a grid framework. To the best of our knowledge, Odyssey is the first system to use history-based prediction to model resource demand as a function of fidelity in adaptive applications.

6 Conclusion

We have shown in this paper that multi-fidelity computation supported by predictive resource management can improve performance in mobile interactive applications. Our performance evaluation shows that

- We reduce mean latency by 60% and variability by 30% for GLVU subjected to a time-varying load.
- History-based demand prediction is accurate and effective, with prediction errors as low as 0.3% for some resources and never higher than 24% in our case studies.
- The cost of using Odyssey is modest, involving 500–1000 additional lines of code per application and 20 ms of runtime overhead per interactive operation.

Throughout the paper, we have indicated areas for incremental improvement; here we mention a few medium to long-term goals for future research. We would like to test Odyssey with a full-fledged AR application on wearable hardware, including location tracking and machine vision as well as rendering, and 100 ms latency bounds rather than 1 s. We would like to further automate the construction of demand predictors: for example, by building platform-independent CPU predictors that can be used across processor architectures. We would like to combine demand prediction with QoS-based allocation such that the system can simultaneously optimize allocation across, and adaptation within, applications [31]. Finally, we would like to explore *mixed-initiative* [20] approaches that combine direct user modification of utility functions with automated inference by the system about user preferences.

References

[1] T. F. Abdelzaher. An Automated Profiling Subsystem for QoS-Aware Services. In *Proc. 6th IEEE Real-Time Technology and Applications Symposium (RTAS '00)*, pages 208–217, Washington, DC, June 2000.

[2] A. Arpaci-Dusseau and R. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, pages 43–56, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[3] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. MacIntyre. Recent Advances in Augmented Reality. *IEEE Computer Graphics and Applications*, 21(6):34–47, Nov./Dec. 2001.

[4] R. K. Balan, M. Satyanarayanan, S. Park, and T. Okoshi. Tactics-Based Remote Execution for Mobile Computing. In *Proc. 1st International Conference on Mobile Systems, Applications, and Services (MobiSys '03)*, San Francisco, CA, May 2003.

[5] M. Billinghurst, S. Weghorst, and T. A. Furness. Wearable Computers for Three Dimensional CSCW. In *Proc. International Symposium on Wearable Computers*, pages 39–46, Cambridge, MA, Oct. 1997.

[6] S. Chandra and C. Ellis. JPEG Compression Metric as a Quality Aware Image Transcoding. In *Proc. 2nd USENIX Symposium on Internet Technologies and Systems (USITS '99)*, pages 81–92, Boulder, CO, Oct. 1999.

[7] S. Chandra, C. Ellis, and A. Vahdat. Differentiated Multimedia Web Services Using Quality Aware Transcoding. In *Proc. 19th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings (INFOCOM '00)*, pages 961–969, Tel Aviv, Israel, Mar. 2000.

[8] M. F. Cohen and J. R. Wallace. *Radiosity and Realistic Image Synthesis.* Academic Press Professional, Boston, MA, 1993.

[9] E. de Lara, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based Adaptation for Mobile Computing. In *Proc. 3rd USENIX Symposium on Internet Technologies and Systems (USITS-01)*, pages 159–170, Berkeley, CA, Mar. 2001.

[10] P. A. Dinda. Online Prediction of the Running Time of Tasks. In *Proc. 10th IEEE International Symposium on High Performance Distributed Computing (HPDC '01)*, pages 383–394, San Francisco, CA, Aug. 2001.

[11] A. H. Dutoit, O. Creighton, G. Klinker, R. Kobylinski, C. Vilsmeier, and B. Bruegge. Architectural Issues in Mobile Augmented Reality Systems: a prototyping case study. In *Proc. Eighth Asian Pacific Conference on Software Engineering (APSEC'2001)*, pages 341–344, Macau, China, Dec. 2001.

[12] S. Feiner, B. MacIntyre, T. Höllerer, and A. Webster. A Touring Machine: Prototyping 3D Mobile Augmented Reality Systems for Exploring the Urban Environment. In *Proc. International Symposium on Wearable Computers*, pages 74–81, Cambridge, MA, Oct. 1997.

[13] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-Tuned Remote Execution for Pervasive Computing. In *Proc. 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 61–66, Schloss Elmau, Germany, May 2001.

[14] J. Flinn, S. Park, and M. Satyanarayanan. Balancing Performance, Energy, and Quality in Pervasive Computing. In *Proc. 22nd International Conference on Distributed Computing Systems (ICDCS '02)*, pages 217–226, Vienna, Austria, July 2002.

[15] J. Flinn and M. Satyanarayanan. Energy-Aware Adaptation for Mobile Applications. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 48–63, Kiawah Island, SC, Dec. 1999.

[16] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*, pages 160–170, Cambridge, MA, Oct. 1996.

[17] M. Garland and P. S. Heckbert. Surface Simplification Using Quadric Error Metrics. In *Proc. SIGGRAPH '97*, pages 209–216, Los Angeles, CA, Aug. 1997.

[18] C. F. Gauss. *Theoria Combinationis Observationum Erroribus Minimum Obnoxiae.* Royal Society of Göttingen, 1821.

[19] M. Harchol-Balter and A. B. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. In *Proc. Joint International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS '94)*, pages 13–24, Nashville, TN, May 1994.

[20] E. Horvitz. Principles of Mixed-Initiative User Interfaces. In *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*, pages 159–166, Pittsburgh, PA, May 1999.

[21] N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive Application-Performance Modeling in a Computational Grid Environment. In *Proc. 8th IEEE International Symposium on High Performance Distributed Computing (HPDC '99)*, pages 47–54, Los Angeles, CA, Aug. 1999.

[22] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A Scalable Solution to the Multi-Resource QoS Problem. In *Proc. 20th IEEE Real-Time Systems Symposium (RTSS '99)*, pages 315–326, Phoenix, AZ, Dec. 1999.

[23] W. E. Leland and T. J. Ott. Load-balancing Heuristics and Process Behavior. In *Proc. Joint International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS '86)*, pages 54–69, Raleigh, NC, May 1986.

[24] R. B. Miller. Response Time in Man-Computer Conversational Transactions. *AFIPS Fall Joint Computer Conference Proceedings*, 33:267–277, Dec. 1968.

[25] D. J. Musliner, E. H. Durfee, and K. G. Shin. Any-Dimension Algorithms. In *Proc. 9th IEEE Workshop on Real-Time Operating Systems and Software (RTOSS '92)*, pages 78–81, May 1992.

[26] K. Nahrstedt, D. Xu, D. Wichadukul, and B. Li. QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments. *IEEE Communications*, 39(11):140–148, Nov. 2001.

[27] D. Narayanan. *Operating System Support for Mobile Interactive Applications.* PhD thesis, Carnegie Mellon University, Aug. 2002.

[28] D. Narayanan, J. Flinn, and M. Satyanarayanan. Using History to Improve Mobile Application Adaptation. In *Proc. 3rd IEEE Workshop on Mobile Computing Systems and Applicatons*, pages 31–40, Monterey, CA, Dec. 2000.

[29] C. Narayanaswami, N. Kamijoh, M. Raghunath, T. Inoue, T. Cipolla, J. Sanford, E. Schlig, S. Venkiteswaran, D. Guniguntala, V. Kulkarni, and K. Yamazaki. IBM's Linux watch, the challenge of miniaturization. *IEEE Computer*, 35(1):33–41, Jan. 2002.

[30] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 276–287, Saint Malo, France, Oct. 1997.

[31] D. Petrou and D. Narayanan. Position Summary: Hinting for Goodness' Sake. In *Proc. 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, page 177, Schloss Elmau, Germany, May 2001.

[32] M. Satyanarayanan and D. Narayanan. Multi-Fidelity Algorithms for Interactive Mobile Applications. *Wireless Networks*, 7:601–607, 2001.

[33] B. S. Siegell and P. Steenkiste. Automatic Generation of Parallel Programs with Dynamic Load Balancing. In *Proc. 3rd IEEE International Symposium on High Performance Distributed Computing (HPDC '94)*, pages 166–175, San Francisco, CA, Aug. 1994.

[34] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A Feedback-Driven Proportion Allocator for Real-Rate Scheduling. In *Proc. 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 145–158, New Orleans, LA, Feb. 1999.

[35] The Walkthru Project. GLVU source code and online documentation. **http://www.cs.unc.edu/~walk/software/glvu/**, Feb. 2002.

[36] G. K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–44, Apr. 1991.

[37] A. Webster, S. Feiner, B. MacIntyre, W. Massie, and T. Krueger. Augmented Reality in Architectural Construction, Inspection and Renovation. In *Proc. ASCE Third Congress on Computing in Civil Engineering*, pages 913–919, Anaheim, CA, June 1996.

[38] A. J. Willmott. Radiator source code and online documentation. **http://www.cs.cmu.edu/~ajw/software/**, Oct. 1999.

[39] P. Young. *Recursive Estimation and Time-Series Analysis.* Springer, 1984.

Host Mobility Using an Internet Indirection Infrastructure

Shelley Zhuang   Kevin Lai   Ion Stoica   Randy Katz   Scott Shenker*
University of California, Berkeley
{*shelleyz, laik, istoica, randy*}@*cs.berkeley.edu*

Abstract

We propose the Robust Overlay Architecture for Mobility (ROAM) to provide seamless mobility for Internet hosts. ROAM is built on top of the Internet Indirection Infrastructure ($i3$). With $i3$, instead of explicitly sending a packet to a destination, each packet is associated with an identifier. This identifier defines an *indirection* point in $i3$, and is used by the receiver to obtain the packet.

ROAM takes advantage of end-host ability to *control* the placement of indirection points in $i3$ to provide efficient routing, fast handoff, and preserve location privacy for mobile hosts. In addition, ROAM allows end hosts to move simultaneously, and is as robust as the underlying IP network to node failure. We have developed a user-level prototype system on Linux that provides transparent mobility without modifying applications or the TCP/IP protocol stack. Simulation results show that ROAM's latency can be as low as 0.25-40% of Mobile IP. Experimental results show that with soft handoff the TCP throughput decreases only by 6% when there are as many as 0.25 handoffs per second.

1 Introduction

While the wired Internet reaches many homes and businesses, the wireless Internet has the potential to not just reach, but encompass all the spaces that people use to live, work, and travel. Wireless data services (e.g., 802.11b, GPRS, 3G cellular) will soon provide the potential for ubiquitous, though heterogeneous, coverage. To realize this potential, users will want both seamless connectivity (flows uninterrupted by mobility) and continuous reachability (the ability of other hosts to contact the user's host despite mobility). These services would enable users to run applications such as IP telephony, instant messaging, and audio streaming while mobile.

Unfortunately, the standard Internet cannot provide these services. The fundamental problem is that the Internet uses IP addresses to combine the notion of unique host identifier with location in the network topology. For a mobile host to have seamless connectivity and continuous reachability, it must retain its identifier while changing its location. Previous mobility proposals decouple this binding by introducing a fixed indirection point (e.g., Mobile IP [1]), redirecting through the DNS (e.g., TCP Migrate [2]), or using indirection at the link layer (e.g., cellular mobility schemes).

However, these proposals lack one or more of the following properties to fully realize the promise of ubiquitous mobility:

- *Efficient routing*: packets should be routed on paths with latency close to the shortest path provided by IP routing.
- *Efficient handoff*: the loss of packets during handoff should be minimized and avoided, if possible.
- *Fault tolerance*: communication between mobile hosts should not be more vulnerable to faults than communication between stationary hosts.
- *Location privacy*: the host's topological location should not be revealed to other end-hosts.
- *Simultaneous mobility*: end hosts should be able to move simultaneously without breaking an ongoing session between them.
- *Personal/session mobility*: a user should be able to redirect a new session or migrate an active one from one application or device to another one when a better choice becomes available [3, 4, 5].
- *Link layer independence*: users should be able to seamlessly operate across heterogeneous link layer technologies, not all of which support the same link layer mobility scheme (e.g., GSM mobility).

In this paper, we propose (to the best of our knowledge) the first solution to achieve all of these properties. Our solution, called Robust Overlay Architecture for Mobility (ROAM), is built on top of the Internet Indirection Infrastructure ($i3$) [6]. $i3$ is implemented as an overlay network on top of IP, and provides a *rendezvous*-based

*ICSI Center for Internet Research (ICIR), Berkeley, shenker@icsi.berkeley.edu communication abstraction. In $i3$, each packet is sent to an identifier. To receive a packet, a receiver inserts a trigger, which is an association between the packet's identifier and the receiver's address. The trigger is stored at an $i3$ node (server). Each packet is routed through the overlay network until it reaches the $i3$ server which stores the trigger. Once the matching trigger is found the packet is forwarded to the address specified by the trigger. Thus, the trigger plays the role of an *indirection* point that relays packets from the sender to the receiver.

ROAM addresses each of the properties described above. For instance, since an $i3$ identifier can be bound to a host, session, or person (unlike Mobile IP, where an IP address can only be bound to a host), personal/session mobility applications can leverage the ROAM infrastructure for efficiency, fault tolerance, and privacy. Section 4 discusses in detail how ROAM achieves the above properties.

At the architectural level, this paper makes two contributions. First, it demonstrates the benefit of giving end-hosts *control* on the placement of the indirection points. This allows, end-hosts to optimize the routing and handoff efficiency. Second, it demonstrates the benefits of a mobility architecture based on a shared overlay network. Such a solution leverages the robustness of the overlay networks.

In addition, we use a proxy based solution to transparently support unmodified applications on an unmodified Linux kernel. Using our prototype implementation, we show that our solution can perform rapid soft handoffs with no noticeable disruption of TCP throughput.

The paper is organized as follows. Section 2 presents the related work, and Section 3 gives an overview of $i3$. Section 4 discusses the design of ROAM, and Section 5 presents the ROAM support for legacy applications. Section 6 presents some implementation details. Section 7 presents simulation and experimental results. Finally, Section 8 discusses some open issues, and Section 9 concludes the paper.

2 Related Work

In this section we review the main mobility proposals.

Several link layer technologies provide mobility at the link layer (e.g., as in Ricochet [7], 802.11b, or GSM). However, these solutions preclude mobility across link layer technologies. In addition, hiding mobility at the link layer results in a reinvention of mobility support in each new wireless system; solving the mobility problem at the network layer results in a reusable mobility infrastructure for all link technologies.

One proposal to achieve mobility in the Internet is Mobile IP (MIP). MIP in IPv4 [1] and IPv6 [8] uses an explicit indirection point, called the Home Agent (HA), to encapsulate and relay the Correspondent Host's (CH) initial packet to the Mobile Host (MH). MIP provides the following options that determine how the following packets are routed: 1) triangle routing, 2) bidirectional tunneling, and 3) route optimization.

As noted by Cheshire and Baker [9] no MIP routing option is clearly better than the others; instead, different options are suitable for different circumstances. Options (1) and (2) preserve location privacy, but routing can be inefficient when the MH and CH are close relative to their distance from the HA. With route optimization (an extension in MIPv4 [10], but standard in MIPv6), the MH conveys its care-of IP address to the CH using a Binding Update (BU). Routing is efficient because the ratio of the latency of the optimized route to the latency of the shortest IP path (or *latency stretch*) is 1.0. However, the CH must be modified to support MIPv4 with route optimization or IPv6. This also exposes the MH's current care-of address (and therefore its location) to the CH, thus compromising location privacy. In certain delay-sensitive or real-time applications, the latency involved in handoffs can be above the threshold if the MH is far away from the CH.

In general, the dependence in MIP on a fixed HA reduces fault tolerance. If the HA or its network fails or is overloaded, then the MH will be unreachable.

To address routing anomalies and robustness issues associated with a fixed HA, researchers have proposed the notion of dynamic home agents in MIPv4 [11]. However, the actual algorithm used to discover and allocate a nearby home agent is still under investigation. MIPv6 provides a *dynamic home agent address discovery* mechanism [8] that allows a MH to dynamically discover the IP address of a HA on its *home network*. This scheme increases the robustness of MIPv6 as the HA is no longer a statically fixed entity, but it does not address routing inefficiencies caused by routing through the HA when the MH is far away from its home network.

Recently, two mechanisms have been proposed to increase handoff performance in MIPv4 and MIPv6: (1) low latency handoff [12], and (2) fast handover [13]. The first mechanism attempts to send a BU in advance of an actual link-layer handoff when the handoff is anticipated. However, timing must be arranged such that the BU completes before the actual handoff does, which may be hard to achieve in practice. Similar in concept to Regionalized Tunnel Management [14] and Hierarchical Mobility [15] extensions in MIPv4 and MIPv6, the second mechanism sets up a bi-directional tunnel between an anchor Foreign Agent (FA) that stays the same during rapid movements and the current FA. This allows the MH to delay a formal BU to the HA which minimizes the impact on real-time applications. However, this mechanism relies on the existence of a FA in *each* network the MH visits. Furthermore, the use of link-layer triggers and inter-FA advertisements in these mechanisms assumes a homogeneous link-layer technology. In contrast to both these mechanisms, ROAM supports fast handoff by giving end-hosts implicit control over trigger placement.

The Host Identity Protocol [16] supports mobility by decoupling the transport from the network layer, and binding the transport to a host identity. Similarly, Location Independent Networking for IPv6 [17] specifies a unique identifier for a host regardless of its location. These ideas are in line with the seminal work by Jerome Saltzer on host identifier and locator separation [18]. However, $i3$ provides a general-purpose indirection infrastructure that enables a variety of communication services beyond mobility, such as multicast, and transcoding. Section 4.2.2 will present the use of $i3$ multicast to support soft handoffs for mobile hosts.

Supporting Mobility for TCP with SIP [19] spoofs constant TCP endpoints in a similar way to MIP with route optimization. This requires modifying the IP stack of the CH.

The Mobility Support using Multicasting in IP (MSM-IP) architecture [20, 21] implements mobility using IP Multicast [22]. The main advantages of MSM-IP are that it can have low latency and do handoffs with little or no packet loss. Several studies [21] [23] [24] have shown that multicast mobility can cut the latency stretch of Mobile IP in half and significantly reduce packet loss due to handoffs. However, the MSM-IP location service is a single point of failure and is vulnerable to overload, network faults, and host faults.

In TCP Migrate [2], both the MH and CH use a modified form of TCP which can tolerate a change in IP address during a connection. The CH uses DNS to learn the current address of the MH, which updates DNS every time it moves. Since TCP Migrate does not use an indirection point, it can achieve an optimal latency stretch of 1.0 and is as fault tolerant as IP routing. However, it lacks simultaneous mobility support, requires modification of the TCP implementations on both the MH and the CH, and does not preserve location privacy. TCP Migrate is well suited for person-to-server applications with short-lived flows like email and web browsing.

The mobility schemes previously described in this section track mobile hosts. In contrast, personal and session mobility schemes (e.g., The Mobile People Architecture (MPA) [4] ICEBERG [5], and Telephony Over Packet networkS [3]) track people or sessions. This allows redi-

$i3$'s Application Programming Interface (API)	
$sendPacket(p)$	send packet
$insertTrigger(t)$	insert trigger
$removeTrigger(t)$	remove trigger

(a)

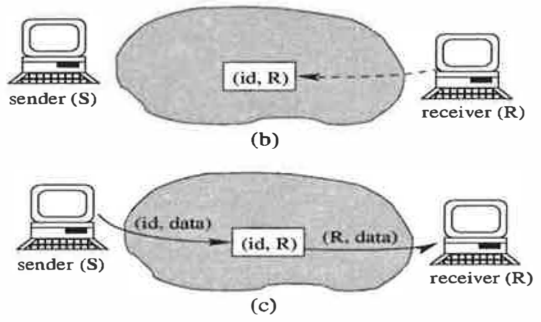


Figure 1: (a) $i3$'s API. Example illustrating communication between two nodes: (b) Receiver R inserts trigger (id, R). (c) Sender S sends packet (id, $data$).

rection of new sessions or migration of active sessions to a completely different application or device according to user connectivity (e.g., which devices are currently accessible to the user) and user preferences (e.g., less expensive or higher performance). In contrast, Mobile IP redirects flows to the same device regardless of whether the user can actually use the device or not. The costs of personal/session mobility schemes are modifications to applications (unlike Mobile IP) and an indirection infrastructure (e.g., the Personal Proxy in MPA).

In contrast to all of the above schemes, the novelty of our approach is the use of an overlay indirection infrastructure that gives endhosts control over the placement of the indirection points. As a result, ROAM achieves efficiency, robustness, location privacy, and simultaneous mobility. In addition, the flexibility of $i3$ identifiers allows ROAM to support mobility at any layer; $i3$ identifiers can be bound to hosts as well as sessions and people.

3 Background

In this section we present a brief overview of an Internet Indirection Infrastructure, $i3$ [6], which forms the foundation for our mobility solution. The purpose of $i3$ is to provide indirection; that is, it decouples the act of sending from the act of receiving. The $i3$ service model is simple: sources send packets to a logical *identifier*, and receivers express interest in packets sent to an identifier. Delivery is best-effort like in today's Internet, with no guarantees about packet delivery.

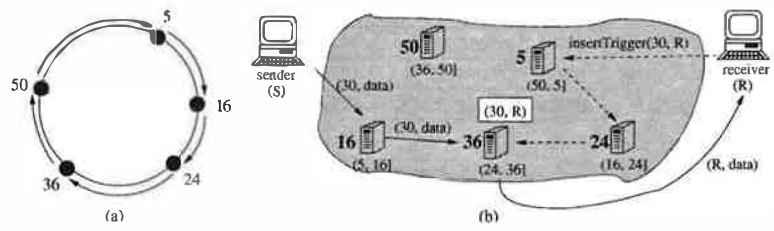


Figure 2: (a) A Chord identifier circle for $m = 6$, with 5 servers identified by 5, 16, 24, 36, and 50, respectively. (b) Receiver R inserting trigger $(30, R)$, and sender S sending packet $(30, data)$.

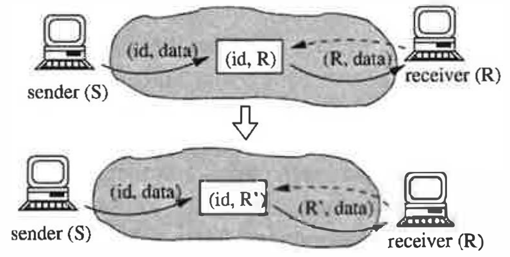


Figure 3: Upon changing its address from R to R', a receiver needs only to update its trigger. This change is transparent to the sender.

3.1 Rendezvous-based Communication

The service model is a rendezvous-based communication abstraction. In its simplest form, a packet is a pair $(id, data)$ where id is an m-bit identifier[1] and $data$ is the payload (typically a normal IP packet payload). Receivers use *triggers* to indicate their interest in packets. In its simplest form, a trigger is a pair $(id, addr)$, where id is the trigger identifier, and $addr$ is a node's address, consisting of an IP address and UDP port number. A trigger $(id, addr)$ indicates that all packets sent to identifier id should be forwarded (at the IP layer) by the $i3$ infrastructure to the node with address $addr$. More specifically, the rendezvous-based communication abstraction exports the three primitives shown in Figure 1(a).

Figure 1(b) illustrates the communication between two nodes, where receiver R wants to receive packets sent to id. R inserts the trigger (id, R) into the network. When a packet is sent to identifier id, the trigger causes it to be forwarded via IP to R.

Thus, as in IP multicast, identifier id represents a logical rendezvous between the sender's packets and the receiver's trigger. This level of indirection decouples the sender from the receiver and enables them to be oblivious to the other's location. However, unlike IP multicast, hosts in $i3$ are free to place their triggers. This can alleviate the triangle routing problem in Mobile IP. In addition, $i3$ can be generalized to support multicast, anycast, and service composition. For more details refer to [6].

3.2 $i3$ Implementation

$i3$ is implemented as an overlay network composed of servers that store triggers and forward packets.

To maintain this overlay network and to route packets in $i3$, we use the Chord lookup protocol [25]. Chord assumes a circular identifier space of integers $[0, 2^m)$, where 0 follows 2^m 1. Every $i3$ server has an identifier in this space, and all trigger identifiers belong to the same identifier space. The $i3$ server with identifier n is responsible for all identifiers in the interval $(n_p, n]$, where n_p is the identifier of the node preceding n on the identifier circle. Figure 2(a) shows an identifier circle for $m = 6$. There are five $i3$ servers in the system with identifiers 5, 16, 24, 36, and 50, respectively. All identifiers in the range (5, 16] are mapped on server 16, identifiers in (17, 24] are mapped on server 24, and so on.

When a trigger $(id, addr)$ is inserted, it is stored at the $i3$ node responsible for id. When a packet is sent to id, it is routed by $i3$ to the node responsible for its id; there it is matched against (any) triggers for that id and forwarded (using IP) to all hosts interested in packets sent to that identifier. Chord ensures that the server responsible for an identifier is found after visiting at most $O(\log n)$ other $i3$ servers irrespective of the starting server (n represents the total number of servers in the system). To achieve this, Chord requires each node to maintain only $O(\log n)$ routing state. Chord allows servers to leave and join dynamically, and it is highly robust against failures. For more details refer to [25]. Figure 2(b) shows an example in which trigger $(30, R)$ is inserted at node 36 (i.e., the node that maps (24, 36], and thus is responsible for identifier 30). Packet $(30, data)$ is forwarded to server 30, matched against trigger $(30, R)$, and then forwarded via IP to R.

Note that packets are not stored in $i3$; they are only forwarded. End hosts must periodically refresh their triggers in $i3$. Hosts need only know one $i3$ node to use the $i3$ infrastructure. This can be done through a static configuration file, or by a DNS lookup assuming $i3$ is associated with a DNS domain name. In Figure 2(b), the sender knows only server 16, and the receiver knows only server 5.

[1]In the implementation presented in this paper, we use $m = 256$. Such a large value of m allows end hosts to choose trigger identifiers independently since the chance of collision is minimal. In addition, a large m makes it very hard for an attacker to guess a particular trigger identifier.

4 ROAM Design

In this section, we describe ROAM, which provides an end-to-end architecture for Internet host mobility on top of $i3$.

Achieving host mobility using $i3$ is straightforward. A mobile host that changes its address from R to R' as a result of moving from one subnetwork to another can preserve end-to-end connectivity by simply updating each of its existing triggers from (id, R) to (id, R'), as shown in Figure 3.

ROAM exhibits the following desirable properties:

Efficient routing. ROAM takes into account the mobility pattern of the end-hosts to improve $i3$'s ability to provide low latency stretch (see Section 4.1).

Efficient handoff. ROAM implements fast handoff and multicast-based handoff to reduce packet loss during handoffs (see Section 4.2).

Fault tolerance. Since triggers are periodically refreshed, ROAM recovers gracefully from server failure. If a server fails, the triggers stored at that server are inserted at another server the next time they are refreshed.[2] Section 7.1.4 evaluates ROAM's robustness through simulation.

Simultaneous mobility. The sending and receiving hosts can move simultaneously while $i3$ serves as an anchor point for the two sides of the communication channel.

Location privacy. ROAM allows end-hosts the flexibility of choosing triggers to not reveal any location information. Section 4.3 discusses the tradeoffs between location privacy and routing efficiency.

Personal/session mobility. Unlike Mobile IP, ROAM allows a user to redirect a new session or migrate an active one from one application or device to another when a better choice becomes available (see Section 4.4).

Next, we describe efficient routing, efficient handoff, location privacy, and personal/session mobility in more detail. The key to achieving these properties is the ability of end-hosts to *control* the location of a trigger in $i3$. Since fault tolerance and simultaneous mobility follow directly from $i3$'s properties, we do not discuss them here.

4.1 Efficient Routing

Although the Chord lookup protocol limits the number of hops traversed in $i3$ to $O(\log n)$, the delay on each hop may be comparable or even larger than the IP shortest path between the MH and the CH. This can result in unacceptably high delay. To deal with this problem, $i3$ uses two techniques: (1) trigger server caching, and (2) trigger sampling [6]. With the first technique, an end-host caches the server storing a particular trigger, and then sends trigger refresh messages and data packets matching the trigger to that server directly via IP. For example, in Figure 2, both the sender (S) and the receiver (R) would cache server[3] 36.

While caching ensures that subsequent packets will traverse only one $i3$ server, the delay can still be large if that server is far from both end-hosts. To address this problem, end-hosts use trigger sampling to pick triggers stored at nearby servers. An end-host picks triggers with random identifiers, measures the round trip delay to the servers that store those triggers, and then uses the trigger with the lowest delay. Note that an end-host only needs to sample at most once per location change, and not every time it opens a connection. As shown in [6], this method is quite efficient. In an $i3$ system with 2^{16} servers, taking only 32 samples results in a 90th percentile latency stretch of only 1.5.

Next, we present an extension of these techniques that takes into account the movement pattern of mobile hosts.

4.1.1 Mobility-Aware Trigger Caching

We assume that mobile hosts are likely to move in a pattern where some moves are short (in geographic distance and network latency), but some moves are very far [26]. This pattern corresponds to a person who drives around a metropolitan area which is a few 10's of miles in diameter, but occasionally flies hundreds or thousands of miles to another location. This pattern also fits a user who moves among different network technologies with widely varying network latency.

We cache sampled triggers to take advantage of this pattern. The goal is to create diversity in the cache so that a trigger in the cache is near each of the remote locations that a mobile host visits (perhaps infrequently), while preventing the frequent local moves from polluting the cache. When the mobile host changes its network address, it randomly samples $i3$ servers as described above, caches the result, and measures the delay to every trigger in the cache. When the cache is full, and the new sample is closer than any in the cache, then we must select a cache entry to evict. If the new sample is much closer than the next closest cache entry (e.g., the new sample's latency is less than 50% of the latency of lowest latency cache entry), then we replace the least recently used trigger in the cache. That the new sample is much closer than the next closest sample indicates that the mobile host is

[2] To make $i3$ server failure completely transparent to end-hosts, $i3$ can replicate triggers [6].

[3] Since the trigger can be reused across connections, the $O(\log n)$ traversal only needs to be done when $i3$ servers fail or when using a trigger for the first time.

probably at a location that is far from any it has visited before, so we evict the entry we are least likely to use again. If instead the new sample is not much closer than the next closest entry in the cache is (e.g. the new sample's latency is 50%-100% of the latency of the next closest cached trigger), then we replace that entry with the new sample. This indicates that the mobile host is relatively close to a recently visited location, and the new sample is a better server for that location.

In Section 7.1, we show by simulation that this caching scheme can reduce the latency stretch to nearly 1.0.

4.2 Efficient Handoff

To reduce the loss of packets during handoffs, ROAM supports fast handoff and multicast-based handoff.

4.2.1 *Fast Handoff*

Existing mobility systems such as Mobile IP or TCP Migrate propagate address binding updates (BUs) all the way to a HA or CH. As a result, a potentially large number of packets may be in flight when the path latency from the MH to the HA or CH is high. If the MH stops receiving packets at the old IP address before starting to receive packets at the new address (*cold switch*), then those in-flight packets will be lost.

With ROAM, end-hosts can alleviate this problem by choosing indirection points (i.e., triggers) that map onto nearby $i3$ servers. Since the number of packets that are lost during a cold-switch is proportional to the delay between the MH and the indirection point, this choice will reduce packet loss. In Section 7.2.2, we use experiments to compare the performances of ROAM and MIPv6 with cold-switching. See section 2 for a qualitative comparison of our approach to two recently proposed mechanisms to increase the performance of handoff in MIPv4 and MIPv6 [12, 13].

4.2.2 *Multicast-based Soft Handoff*

When a MH moves from one network to another, there may be an interval during which it has poor connectivity (either lost packets or low bandwidth) in the new network, but good connectivity in the old network. If the MH performs handoff too early, then its performance can suffer from poor connectivity in the new network. On the other hand, if the MH performs handoff too late, then it may lose packets as the connectivity in the old network degrades.

The solution in ROAM is to use the generalized level of indirection provided by $i3$ to do multicast-based soft handoff. In the situation described above, when the MH can obtain an address in the new network, the MH's proxy inserts a trigger with the same identifier as its existing trigger, but associated with the new address. This causes the same packets to be delivered to both the old and new addresses. This allows the MH to take advantage of the best available connectivity. There are two things worth noting. First, the use of multicast is completely transparent to the sender. Second, fast handoff is still necessary for cases when the MH cannot listen simultaneously at both addresses. For example, an 802.11b client cannot be simultaneously connected to two base stations on different channels [27]. We address the problems of determining when to stop using multicast and how to suppress duplicate packets in Section 6.1. We discuss the implication of multicast on communication privacy in Section 8.

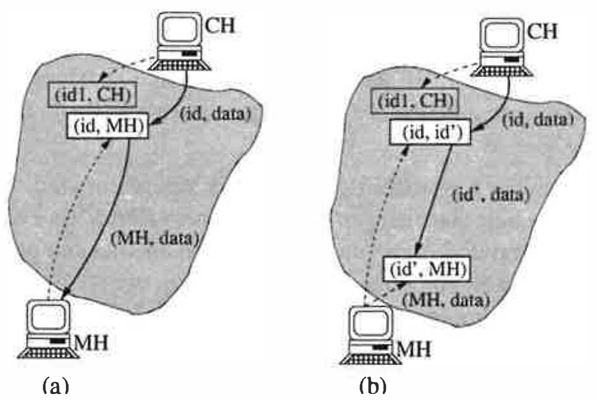


Figure 4: Achieving location privacy: (a) MH chooses a trigger close to the CH; (b) MH uses two triggers to preserve fast handoff.

4.3 Location Privacy

While choosing a trigger (id, MH) at an $i3$ server close to the MH improves the routing and handoff efficiency, this choice would reveal (to some extent) the location of the MH. To avoid this problem, the MH can choose id such that the trigger is stored at an $i3$ server close to the CH instead of itself. This would result in a low latency stretch without compromising the MH's location privacy. Let (id_1, CH) be the trigger advertised by the CH to the MH (see Figure 4(a)). Assuming that the CH chooses this trigger close to itself, the MH can simply choose id to share the first 128 bits with id_1. This is because with $i3$, all triggers whose identifiers share the same 128-bit prefix are stored at the same $i3$ server [6].

To also allow fast handoff, the MH can use two triggers, one of the form (id, id') [4] inserted near the CH, and one of the form (id', MH) inserted close to itself (see Fig-

[4]Note this is a generalized form of triggers, which allows a trigger to send a packet to another identifier rather than to an address.

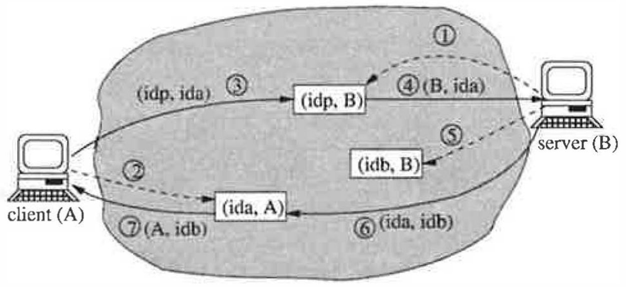


Figure 5: Example of setting up a connection via private triggers id_a and id_b between two hosts A and B. id_p is B's public trigger.

ure 4(a)). Note that this change is transparent to the CH, i.e., the CH will still send packets of the form $(id, data)$ to the MH. Because the CH does not need to know id', the location privacy of the MH is ensured. Moreover, the choice of id and id' ensures a low latency stretch, and enables the MH to do fast handoff by updating trigger (id', MH).

If both end-points require location privacy, they can choose completely random $i3$ servers. The flexibility of $i3$ allows each application to make the tradeoff between location privacy and routing efficiency as desired.

4.4 Personal/Session mobility

While the focus of this paper is on using ROAM for network layer mobility, ROAM also provides support for personal/session mobility (Section 2). Personal/session mobility requires tracking the set of active devices for a user and routing to the optimal device. Devices register themselves for a particular person whenever they detect an authorized user is nearby (e.g., devices have Bluetooth transceivers and users carry Bluetooth smart cards) [28]. Devices register by setting a trigger with an identifier representing that user. Given multiple simultaneously registered devices, the devices follow an agreement protocol to decide which one will handle a particular session (e.g., the least expensive one or the highest performing one). Leveraging the ROAM infrastructure for personal/session mobility removes the cost of deploying infrastructure specific to any particular application or personal/session mobility scheme.

5 Application Support

One of the central goals of ROAM is to support legacy applications. Ideally, this would allow us to transparently run existing applications on top of ROAM. In this section, we first describe how $i3$ can support native applications, and then present our solution for legacy applications.

Notation	Definition
$X.hip$	home IP address of host X
$X.cip$	current IP address of host X
$C.port$	port associated to client process C
$i3\_hdr$	$i3$ packet header (see Figure 7)
$i3\_hdr.id$	$i3$ packet's identifier
$proxy\_hdr$	$i3$ proxy header
$proxy\_hdr.flags$	flags: ID_MASK specifies that $proxy\_hdr$ has an $i3$ identifier. DATA_MASK specifies that the payload has an IP packet.
$H()$	well-known hash function; used to compute public trigger identifier for X as $H(X.hip)$
$trans\_table$	translation table maintained by each $i3$ proxy; each entry is a pair (IP address, $i3$ identifier)

Table 1: Notations used in Section 5.2.

5.1 Support for Native Applications

So far we have assumed that each end-host is free to choose its triggers independently. The natural question is how does an end-host learn about the trigger of another end-host? To answer this question, $i3$ introduces the concept of *public* and *private* triggers [6]. Private triggers are secretly chosen by the application end-points. Public triggers can be computed by all end-hosts in the system and are used to establish initial contact with the desired end-host. For example, the public trigger of the "New York Times" web server can be the hash of its name.

Consider the application in Figure 5 where a client A accesses a web server B. The web server B maintains a public trigger with identifier id_p in $i3$ (step 1). The control path operations are as follows. Client A inserts a private trigger with identifier id_a into $i3$ (step 2), and sends id_a to web server B via B's public trigger id_p (step 3). B receives id_a from $i3$ (step 4) and inserts a private trigger with identifier id_b into $i3$ (step 5). B then sends id_b to A via A's private trigger id_a (step 6), and A receives it from $i3$ (step 7). Finally, data packets from A to B flow through B's private trigger id_b, and through A's private trigger id_a in the reverse direction. The important point to note here is that end-hosts have full control on selecting their private triggers.

5.2 Support for Legacy Applications

Although achieving host mobility for $i3$ native applications is straight-forward, many legacy applications will remain $i3$/ROAM unaware. In designing a solution for these applications, our primary goals are to remain transparent to both applications and the TCP/IP protocol stack. The main host modification required for legacy applications is a user-level ROAM proxy. The proxy serves the following functions: (1) encapsulates and decapsulates IP packets within $i3$ packets, (2) determines the trig-

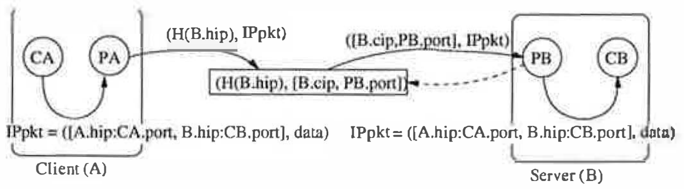

CA
PA
(H(B.hip), IPpkt)
([B.cip,PB.port], IPpkt)
(H(B.hip), [B.cip, PB.port])
PB
CB
IPpkt = ([A.hip:CA.port, B.hip:CB.port], data)
IPpkt = ([A.hip:CA.port, B.hip:CB.port], data)
Client (A)
Server (B)

Figure 6: Legacy application support. $A.hip$ and $B.hip$ represents home IP addresses of hosts A, and B, respectively. Each host has a proxy that intercepts applications packets and send them via $i3$.

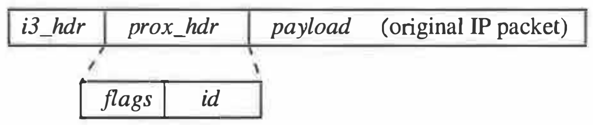


Figure 7: The format of the $i3$ packet handled by the proxy. The fields are explained in Table 1.

gers of remote hosts, and (3) sends the local private trigger to remote hosts. Table 1 gives the notations used in this section.

We assume that each host X has a current IP address denoted by $X.cip$ and a home IP address (e.g., the address of the host in its home network) denoted by $X.hip$. The home address is stored in the end-host's DNS record, and it is used as a source address for all packets sent by legacy applications on X. Each host X runs a ROAM proxy PX that maintains a public trigger $(id, addr)$ where id is computed as a hash on X's home IP address, and $addr$ contains the current address of X and PX's port number, i.e., $[X.cip, PX.port]$. The proxy is responsible for updating the trigger every time the host's current IP address changes.

Figure 6 shows a typical data path in a legacy application, where a client CA running on host A is accessing a web server CB running on host B. (Figure 8 shows the pseudo-code executed by an ROAM proxy.) The source and the destination addresses in the headers of the packets sent by CA are the host IP addresses of A and B, respectively. Upon capturing the packet, PA encapsulates it in $i3$ and proxy headers and sends it to CB through $i3$ using UDP. [5] The identifier of the packet is set to B's public trigger identifier, i.e., $H(B.hip)$ (see function **ip_receive** in Figure 8). The format of the packets handled by $i3$ proxies is shown in Figure 7

When this packet arrives at B (see **i3_receive**), B's proxy (PB) strips off the $i3$ and proxy headers and forwards the packet to the local application. In addition, PB checks to see if the packet is addressed to its own public trigger. If it is, then PB knows that A's proxy (PA) does not have a private trigger for B, so PB should send one. As an optimization, PB sets a timeout to see if it can piggyback the trigger on a packet sent from B's application (CB). Otherwise, when the timeout expires, B's proxy sends the private trigger in a separate packet. An end-host chooses private triggers on a per flow or a per communication peer basis. This precludes a malicious end-host from learning the private trigger used by (the flows of) another end-host and eavesdropping on it.

Assume that CB does send a packet before the timeout expires, then PB piggybacks B's local private trigger on the outgoing packet to A. Since, PB does not know A's private trigger, it uses A's public trigger (as $H(A.hip)$). When PA receives this packet, it inserts B's private trigger into its translation table with $B.hip$ as the key. In addition, PA sees that the packet was sent to its own public trigger, so it also sets a timeout and tries to piggyback its private trigger to B.

When A changes its IP address from $A.cip$ to $A.cip'$ as a result of moving from one subnetwork to another, PA will insert a trigger containing the new IP address $A.cip'$ into $i3$ and remove the trigger containing the old IP address $A.cip$. The trigger identifier itself remains the same. Effectively, host mobility is masked by the $i3$ network from the communicating peer, and end-to-end connectivity is preserved.

While each end-host initially chooses its private triggers such that they are stored on nearby servers, end-hosts may eventually move far from those servers. To address this problem, each end-host can re-sample trigger servers either periodically or once it notices that its current private triggers are experiencing a high latency. The new private triggers can be exchanged using a mechanism identical to the one used to exchange the original private triggers via the public triggers. The only change occurs in the **i3_receive** function: in addition to comparing the packet identifier to the the host's public trigger, we also compare it to the previous private trigger identifier, and then send out the new private trigger if necessary. This operation will be transparent to applications.

6 Implementation Details

The ROAM user-level proxy translates between existing Internet packets and $i3$ packets, and inserts/refreshes triggers on behalf of the applications. Applications do *not* need to be modified, and are unaware of the ROAM proxy. The ROAM proxy uses a virtual link-level interface (similar to [29]), called TUN[6], to transparently cap-

[5]In order to avoid fragmentation due to the encapsulation, the maximum segment size (MSS) TCP header option in a SYN packet is decremented accordingly.

[6]The TUN virtual interface is implemented by the Universal TUN/TAP driver, which is included as a standard feature of the

```
// on receiving an IP packet $p_{ip}$ from local applications
ip_receive($p_{ip}$)
  $p$ = i3_pkt_new();
  $p.payload = p_{ip}$;
  $p.prox\_hdr.flags = p.prox\_hdr.flags \vee$ DATA_MASK;
  // do we need to send a private trigger to the sender?
  if (exist_timeout($p_{ip}.dst\_addr$))
    $p.prox\_hdr.flags = p.prox\_hdr.flags \vee$ ID_MASK;
    $p.prox\_hdr.id = choose\_private\_trigger\_id(p_{ip}.dst\_addr)$;
    timeout_remove($p_{ip}.dst\_addr$);
  $p.i3\_hdr.id$ = i3_id($p_{ip}.dst\_addr$);
  i3_send($p$);

// timeout set by i3_receive for addr has expired
timeout($addr$)
  $p$ = i3_pkt_new();
  $p.prox\_hdr.flags = p.prox\_hdr.flags \vee$ ID_MASK;
  $p.prox\_hdr.id = choose\_private\_trigger\_id(p_{ip}.dst\_addr)$;
  $p.i3\_hdr.id$ = i3_id($p_{ip}.dst\_addr$);
  i3_send($p$);
```

```
// on receiving an i3 packet p from network
i3_receive($p$)
  $p_{ip} = p.payload$; // get encapsulated IP packet carried by
  // does p carry sender's private trigger?
  if ($p.prox\_hdr.flags \wedge$ ID_MASK)
    update($trans\_table, p_{ip}.src\_addr, p.prox\_hdr.id$);
  else refresh($trans\_table, p_{ip}.src\_addr$);
  // was p sent to the local host's public trigger?
  if ($p.i3\_hdr.id = H(p_{ip}.dst\_addr)$)
    // p's source may not know our private trigger identifier
    timeout_set($p_{ip}.src\_addr$); // set a timeout to send it
  // does p contain data for a host's client?
  if ($p.prox\_hdr.flags \wedge$ DATA_MASK) ip_send($p_{ip}$);

// return the i3's identifier corresponding to addr
i3_id($addr$)
  // get destination's private trigger from translation table
  if (exist_entry($trans\_table, addr$))
    return get_id($trans\_table, addr$);
  else return $H(addr)$;
```

Figure 8: The pseudo-code executed by the proxy upon receiving packets from another host via *i*3 and from a host's client. The format of packet p handled by the proxy is given in Figure 7. *trans_table* denotes a translation table that stores the association between (1) a host IP address *addr*, and (2) the identifier of the private trigger inserted by the proxy running on host *addr*.

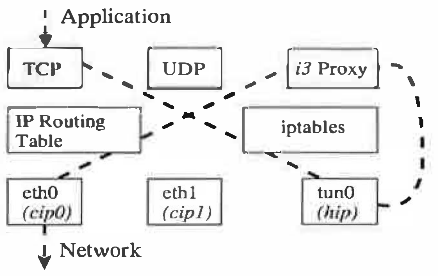


Figure 9: Data link, network, and transport layers on an end-host running the ROAM proxy software. The dashed line shows the path of an outgoing TCP packet.

ture packets at user-level, and to hide host mobility from applications. The TUN interface receives packets from user-level applications instead of from a physical media, and sends them to user-level applications instead of sending packets via physical media.

Users can specify a set of criteria, using the iptables tool, that determines whether a packet is redirected to the TUN virtual interface or passed directly to the IP routing table. For example, if the user specifies the filter "-p udp –dport domain -j ACCEPT", then iptables will pass all DNS query and reply packets directly to the routing table.

Figure 9 illustrates the organization of our software when sending out a packet from the end host. The ROAM proxy reads and translates packets from tun0. To ensure that the translated packet does not get routed to tun0 again, the proxy adds a rule to iptables such that all packets from itself are passed directly to the routing table. Incoming packets from the correspondent host's proxy will arrive at the physical interface and be addressed to the ROAM proxy. The proxy will strip off the *i*3 and proxy headers and send it to TUN, from which the applications will receive the packet (thus taking the reverse of the dashed path shown in Figure 9).

kernel in Linux 2.4 and later.

6.1 Multicast-based Soft Handoff

As a result of multicast-based soft handoff, the *i*3 server will send duplicate encapsulated packets to the MH. To prevent the MH's TCP/IP stack and applications from receiving duplicates of the inner packet (i.e., original IP packet, see Figure 7), the ROAM proxy suppresses duplicates during multicast-based soft handoff.

The ROAM proxy maintains a small window of MD5 [30] digests of recent packets. The proxy computes digests over the first 20 bytes of the IP header and the first 8 bytes of the transport header. The first 28 bytes of a packet are sufficient to differentiate non-identical packets in practice [31]. To minimize duplicates, the window size must be sufficiently large so that a duplicated packet that arrives both via a very low latency link and via a very high latency link will be caught in the window. We use a window size of 1 second, which should be sufficient even if one path is very congested or contains a 500ms satellite link. Note that this scheme detects duplicates received on different addresses, which means that only duplicates generated by $i3$ are eliminated, and *not* duplicate packets sent by the sender (e.g., TCP dup-ack). We show in Section 7 that a TCP bulk transfer flow using multicast-based soft-handoff achieves similar throughput to a flow without mobility.

Another implementation issue is when does the proxy stop using multicast. Our algorithm removes an address when a large fraction of its packets are duplicates as this indicates that the address is redundant. The ROAM proxy maintains a counter of duplicate packets received on both addresses (d) and a counter of packets received on each address (p_i). When $d > \min(p_0/k, p_1/k)$, we simply remove the address that has received fewer packets in the last window. The value $1/k$ is a constant indicating the fraction of an address's packets that must be duplicates before the address can be dropped. In addition, the proxy uses a timeout t to prevent a newly added address with poor connectivity from being removed until the timeout expires. In our implementation, we use $k = 2$ and $t = 30$ s.

7 Evaluation

In this section, we present simulation and experimental results evaluating the benefit and cost of using ROAM.

7.1 Simulations

We use simulation to evaluate ROAM and Mobile IP with triangle routing, bidirectional routing. We show that routing efficiency and fault tolerance are proportional to the amount of mobility infrastructure (either $i3$ servers in $i3$ or Home Agents in Mobile IP) deployed. However, for moderate amounts of infrastructure, ROAM provides higher routing efficiency and fault tolerance than Mobile IP. In addition, ROAM's routing efficiency and fault tolerance scale with the amount of infrastructure devoted to it.

7.1.1 Methodology

We use our own simulator to simulate $i3$ mobility and Mobile IP with its routing options. The simulator is session level, and simulates the creation, maintenance, and measurement of routes in the IP network, Mobile IP, and $i3$. We do not require the level of detail (and consequent overhead) of a packet-level simulator like NS.

Our simulation network topologies consist of three types of nodes: routers, mobility servers ($i3$ servers or HAs), and client hosts (MHs or CHs). We arrange the routers using a transit-stub topology generated with the GT-ITM topology generator [32] with 5000 nodes, where link latencies are 100 ms for intra-transit links, 10 ms for transit-stub links and 1 ms for intra-stub links. In [33], we also present simulations using a power law topology.

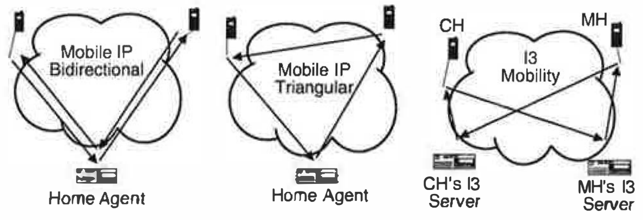


Figure 10: Routing in different mobility schemes.

We define *domain* to be a group of nodes that have low latency links between them. We assume that each router forms its own domain. We consider 5000 possible client hosts, and up to 10000 mobility servers.[7] For a particular topology, we use 50 random choices from the client hosts for the home network (HN). For each choice of HN, we use 2000 random choices from the client hosts for the MH and CH, as described below.

In addition to regular IP routing, we consider three mobility routing schemes: Mobile IP with triangular routing, Mobile IP with bidirectional tunneling, and ROAM (see Figure 10).

With Mobile IP, each MH has a HA associated with it. While the HA is typically assumed to be in the HN, in practice this might be hard to achieve due to deployment costs. In addition, requiring the HA to be in the HN restricts the number of MHs that can be supported. For these reasons, we assume a more incremental deployment model, where a service provider would provide one or more HAs and map multiple users to each one. Therefore, in our simulations, we select the server closest to the HN as the HA.

With ROAM mobility, the MH uses the mobility-aware caching algorithm described in Section 4.1. The MH takes 32 samples in each move, maintains 10 entries in its cache, and replaces close entries when new samples are closer, but not less than 50% closer. These parameters are a compromise between performance and overhead because each sample consumes network bandwidth.

We simulate MH movement according to two mobility models: uniform and Pareto with respect to the HN. Each model defines the distance of the MH from the HN. In the uniform model, the distance of the MH from the HN is uniformly randomly selected from the interval [minimum distance, maximum distance]. In the Pareto home

[7]There is little performance improvement for more than $2N$ servers because at that point, each domain is likely to have a server.

model, the probability that the MH is distance d from the HN is $1/d^2$. This simulates a MH that is close to the HN most of the time, but sometimes moves very far from the HN.

Similarly, we simulate communication with CHs according to three communication models: uniform, Pareto with respect to the HN, and Pareto with respect to the MH's current location in a foreign network. In the uniform model, the CH is uniformly randomly selected from the clients. The Pareto home and Pareto foreign models assign distances to the CH according to the Pareto model given above, but relative to the HN or the MH's current location, respectively. These models simulate a CH that is close to the HN or MH, respectively, most of the time, but is sometimes very far from it.

Given all of these parameters, we measure the round trip time (RTT) of the various mobility schemes as shown in Figure 10. Note that in the ROAM case, both the MH and CH can be mobile, while in the triangular routing and bidirectional tunneling cases, we assume that the CH is stationary (i.e., the CH does not have a HA). If we were to assume that the CH is mobile, then the triangular routing and bidirectional tunneling cases would incur even more latency, so this comparison favors those cases over ROAM.

In all cases, we measure the 90th percentile latency stretch [8] of the various schemes.

7.1.2 *Results: Stretch vs. Infrastructure Size*

Figure 11 shows a series of graphs which compare the 90th percentile stretch of MIP with triangular routing and bidirectional tunneling ("bi"). Each graph shows a different combination of mobility model and communication model.

In the transit-stub network, ROAM matches or exceeds MIP's stretch when more than 1-2% of the transit-stub domains have a server. ROAM matches MIP when one or both of the communication end points (the MH and CH) is close to the HN. We expect that these are the optimal cases for MIP. Indeed, Figures 11 (b), (d), (e), and (f) show that MIP's stretch drops sharply as the number of deployed HAs increases. More HAs increase the likelihood that a HA will be in the HN, thus decreasing the stretch incurred by triangular routing or bidirectional tunneling when the CH and/or MH are close to the HN. However, the figures also show that ROAM's stretch converges with MIP's when more than 50-100 servers are deployed in the network (corresponding to 1-2% of the transit-stub domains having a server). This is because

[8] Calculated as the ratio of the path latency using a particular mobility scheme to the shortest path latency on the underlying network topology.

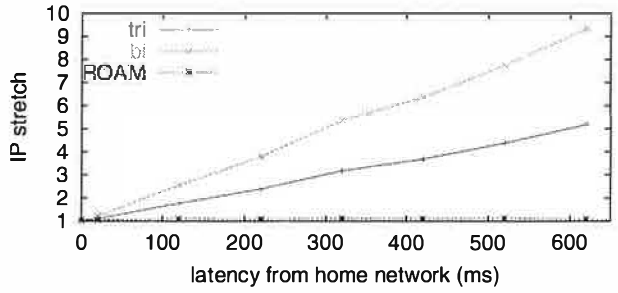


Figure 12: The 90th percentile stretch of MIP (tri/bi) and ROAM.

ROAM is able (through its trigger server caching algorithm) to dynamically find $i3$ servers which are as close to the MH and CH as a statically configured HA.

ROAM significantly improves on MIP's stretch when neither the MH or CH are close to the HN. We expect this to be the worst case for MIP. Figures 11(a) and (c) validate this. Increasing the number of HAs in these cases does not decrease MIP's stretch because having a HA close to the HN does not put it any closer to the CH or MH. In contrast, ROAM's stretch decreases as more servers are deployed because it can still dynamically find closer trigger servers. Figure 11(a) shows that even when the CH, MH, and HN form a triangle with equal distribution of distance on each leg, ROAM's stretch is 40% that of MIP. When the CH and MH are Pareto close (as shown in Figure 11(c)), then ROAM has a stretch 1/400th that of MIP with triangular routing. The difference is so large because the maximum latency in our transit-stub topology is over 1000 ms, while the minimum latency is only 1ms, so the impact of poor routing is very large.

7.1.3 *Results: Stretch vs. Distance from HN*

Figure 12 compares the stretch to the distance of the MH from the HN. As the distance from the HN increases, MIP's stretch increases linearly, while ROAM's stretch remains relatively constant. This simulation uses the transit-stub topology with 10000 mobility servers, a uniform mobility model, and a uniform communication model.

7.1.4 *Results: Fault Tolerance*

In addition to stretch, we also simulate node failures. We vary the failure probability of the clients and servers from 0% to 50% and perform 10,000 runs. In this simulation, we assume that both the MH and CH are mobile and have a HA. We assume that IP routing succeeds when both the MH and CH are operational. We assume that MIP is functional when the MH, CH, MH's HA, and CH's HA are operational. We assume that ROAM is functional

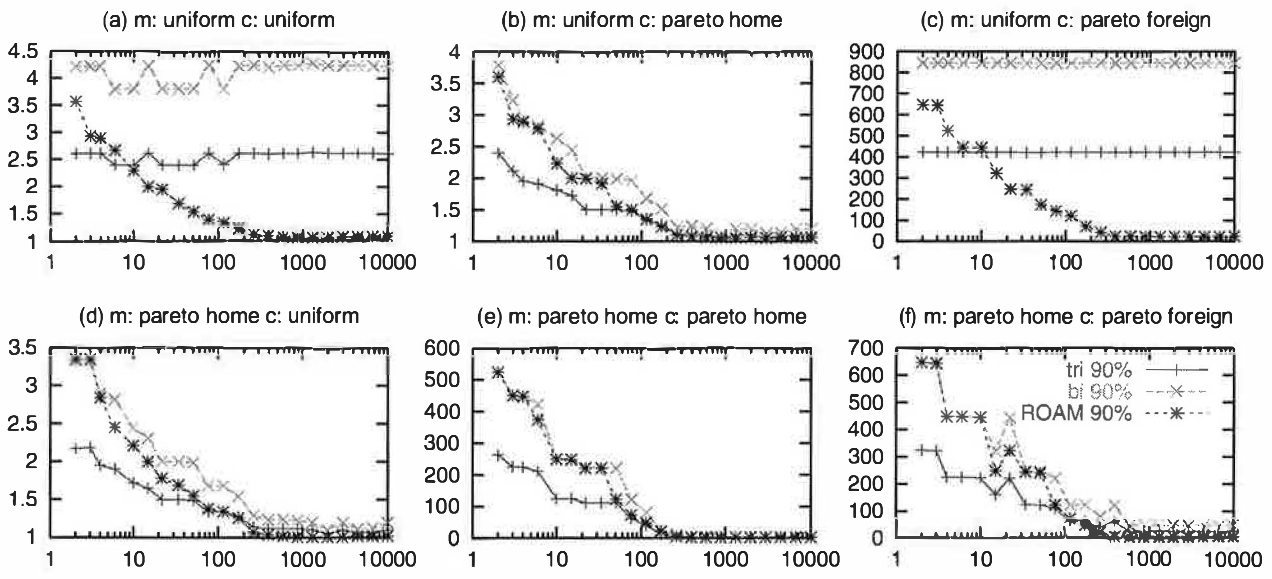


Figure 11: The 90% stretch of MIP with triangular routing ("tri"), bidirectional tunneling ("bi"), and ROAM. In all plots, x-axis represents the number of servers (home agents or $i3$ servers) on a log scale, and y-axis represents the latency stretch. Each plot corresponds to a mobility model ("m") and a communication model ("c"). "uniform", "Pareto home", and "Pareto foreign" indicate how the location is chosen, i.e., according to a uniform distribution, Pareto distribution from home network, and Pareto distribution from the foreign network.

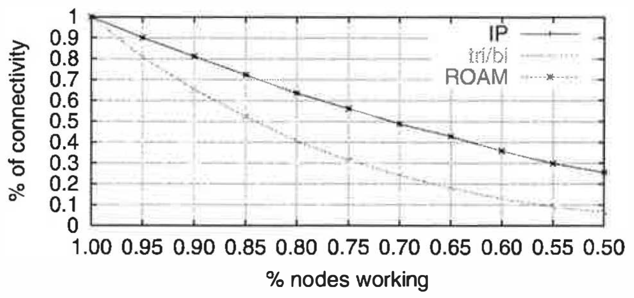


Figure 13: The robustness of IP, MIP (tri/bi), and ROAM.

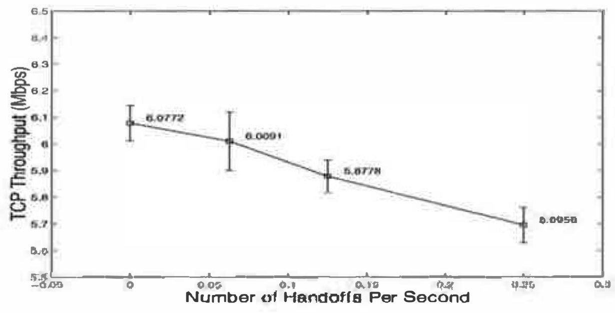


Figure 14: TCP throughput received by the MH as the handoff frequency increases. The vertical error bars show the standard deviations of the receiver throughput.

when the MH and CH are operational, and the MH and CH can both find an operational trigger server in their caches (of size 10).

Figure 13 shows the results of failing nodes on the likelihood of connectivity between the MH and CH. When nodes have a 5% chance of failing, MIP has a 85% likelihood of successful connectivity. When nodes have a 15% chance of failing, MIP likelihood of successful connectivity drops to only 50%. MIP is vulnerable to the failure of the HA's network connectivity. In most cases, a host has only one HA in its HN. As a result, if the HA's network connectivity fails, the MH is unreachable. In contrast, a ROAM host can use any $i3$ server in the Internet. As long as one $i3$ server is operational and the ROAM host has IP connectivity, the host is reachable.

7.2 Experiments

In this section, we describe our test-bed and examine the effect of handoffs on TCP throughput. Our MH is a IBM Thinkpad T23 1.13GHz laptop running Red Hat Linux 7.3 with a 2.4.18 kernel. The CH is a 866 MHz desktop running a 2.4.18 Linux kernel. Our $i3$ server is a 800 MHz desktop running a 2.4.10 Linux kernel.

7.2.1 *Results: Multicast-based Soft Handoffs*

In this experiment, we perform TCP bulk transfers from the CH to the MH. The MH resides in a 10 Mbps Ethernet, and the CH and the $i3$ server reside in a 100 Mbps Ethernet. The MH initiates TCP connections from one location on its subnet, and moves to another location on

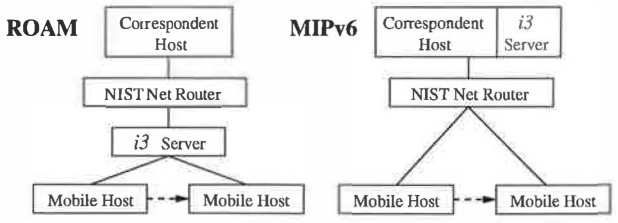


Figure 15: Network topology used for cold switch experiments. As shown by the dashed arrow, the mobile host moves from one location to another on the same subnet during handoff.

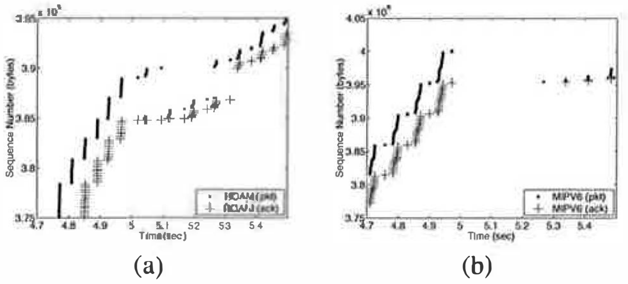


Figure 16: TCP sequence trace showing a bulk transfer with a cold-switch for (a) ROAM, and (b) MIPv6.

the same subnet at a later point, or vice versa. Both MH locations use identical connections with 10Mbps links. The purpose of this simple configuration is to expose the performance impact of multicast-based soft handoffs. Each run involved a TCP bulk transfer lasting 16 seconds and we varied the number of handoffs (0, 1, 2, and 4) performed during each transfer. This was repeated ten times at each handoff frequency. Figure 14 plots TCP throughput and its standard deviation received by MH as the number of handoffs increases during the bulk transfer.

We see that as handoff frequency increases, the TCP throughput degradation is minimal. In fact, there are no losses across the multicast-based soft handoffs as both interfaces are available. The slight performance penalty is caused by the overhead of MD5 digest computation of every packet received and detection of duplicates during handoffs. This demonstrates the effectiveness of ROAM to support rapid handoffs. For example, consider a user on an airplane moving at 540 miles per hour, and cell coverage sizes with diameters of 1.5 miles. In this case, the user makes 6 cell crossings per minute, which can be easily supported by ROAM. To support multiple such users on the airplane, we can use a NAT-like device to aggregate cell-crossings made by users, and thereby alleviate the handoff load on the $i3$ trigger server.

7.2.2 Results: Cold Switch

In this experiment, we compare the handoff performance of ROAM and MIPv6 during a cold switch when the MH is far away from the CH. Figure 15 shows the experimental setup. We use the NIST Net [34] network emulation package to emulate a round trip time (RTT) of 70 ms between the MH and CH. In the setup for ROAM, RTT between the MH and the $i3$ server is approximately 3 ms. The NIST Net router delays packets between the $i3$ server and the CH by 70 ms. We emulate the MIPv6 scenario by running the $i3$ server on the same machine as the CH since binding updates are propagated to the CH in MIPv6. The NIST Net router delays packets between the MH and CH by 70 ms. During a cold switch, the first interface is shutdown around 35-40 ms before the second interface is brought up. During this disconnected interval, t, packets from the $i3$ server to the MH are lost in both ROAM and MIPv6. However, the number of packets that are lost after cold switch completes is proportional to the delay between the MH and the indirection point.

Figure 16 plots the TCP sequence numbers seen at the CH (TCP sender) for the ROAM and MIPv6 scenarios during a cold switch. ROAM recovers from packet loss caused by the cold switch by entering fast retransmit when the MH receives duplicate acknowledgements generated by packets received after the lost packets. However, in MIPv6, the MH loses the entire window of data and the CH waits for a timeout and goes into slow start before retransmitting the lost packets.

If the disconnectivity time due to cold switch is t, and $t < RTT < 2t$, then ROAM can recover by fast retransmit whereas MIPv6 has to recover by timeout. If RTT is greater than $2t$, then both ROAM and MIPv6 can recover through fast retransmit. However, ROAM will recover sooner because of its ability to choose a nearby $i3$ server irrespective of the CH's location, thereby greatly reducing packet loss.

8 Discussion

In this section, we discuss some important security issues and the overhead of ROAM. We then discuss the possibility of using ROAM to exchange only control information, while data packets are forwarded via IP. Finally, we discuss the possibility to replace the ROAM proxy with a NAT-like solution, and some deployment issues.

Eavesdropping. As discussed in Section 4.2.2, $i3$ supports multicast by allowing any host in the network to add a trigger with the same identifier as another host's trigger. However, this allows any host to eavesdrop on another host's communications *if* it knows that host's trigger.

To avoid this problem, $i3$ uses public key cryptography to protect against eavesdropping [6]. When initiating a

Routing	Header Size	Relative Overhead	Transmission Delay
IP	40B	1.25	23ms
Mobile IP	60B	1.88	28ms
ROAM	117B	3.66	41ms
ROAM w/comp	45.8B	1.43	24ms

Table 2: This table shows the total header overhead (including TCP/IP) of various routing schemes. The listed overhead is relative to a 32 byte payload. The transmission delay is for the given header size, a 32 byte payload, and a 32Kb/s link bandwidth.

connection, A encrypts its private trigger id_a under the public key of B, before sending it to B via B's public trigger id_p. Since A's private trigger is encrypted, a malicious user M cannot impersonate B even if it inserts a trigger $(id_p, addr_m)$ into $i3$. A potential disadvantage of this technique is it assumes the existence of a Public Key Infrastructure.[9]

An alternative solution would be to use an EXCLUSIVE_ID flag in the trigger headers to preclude other hosts from inserting triggers with the same identifier. Since private triggers are assumed to be secret, they do not need to have the EXCLUSIVE_ID flag set. This allows applications to use the multicast functionality via private triggers (see Section 4.2.2).

Although setting the EXCLUSIVE_ID flag ensures that no one can eavesdrop on communication destined to a public trigger, an attacker can wait for a host to fail to refresh its public trigger, and insert its own trigger with the same identifier. As a result, all packets destined to that identifier will be received by the attacker. This attack is similar to hijacking a DNS entry. If an end-host wants to eliminate this attack, it may use again cryptography to avoid impersonation by an attacker.

Overhead. Table 2 lists the overhead of various routing schemes. A standard web browser using IP and TCP or an IP telephony application using IP, UDP, and RTP has a total header size of 40 bytes. Mobile IP needs 20 additional bytes for IP in IP encapsulation. The size of $i3$ header in the current implementation is 48 bytes (of which 32 bytes is the $i3$ ID). The proxy header has a minimum size of one byte (see Figure 7). The encapsulating IP and UDP headers total 28 bytes. Thus, the ROAM total header size is 28 (encapsulating packet) + 1 (proxy) + 48 ($i3$ header) + 40 (original packet) = 117. When private IDs are piggybacked in data packets (typically only in the beginning of a connection), the overhead increases by another 32 bytes.

[9]Another possibility would be to use DNS to store public keys, but then ROAM would be as secure as the DNS.

However, header compression can reduce packet header overhead by a factor of 5 [35]. If we compress the 89 bytes of header after the encapsulating header (which must remain uncompressed to route through the Internet), then we reduce the total header size to an average of 45.8 bytes. This only requires modifications to the proxy and $i3$ server software.

Table 2 shows that even for a 32 byte IP telephony payload, the ROAM compressed header overhead is only 18% greater than that of standard TCP/IP. On a hypothetical 5ms latency, 32Kb/s link, the net difference in transmission delay is 5%. This overhead decreases as packet sizes, latencies, and bandwidths increase.

Another source of overhead is the user level proxy which causes each packet to traverse the OS–user level boundary twice. This can reduce the maximum throughput that can be achieved by the end host. However, that maximum is unlikely to be reached even in a relatively high bandwidth wireless network like 802.11b (11Mb/s). If this becomes an issue, the proxy can be eliminated at the cost of implementing its functionality in the kernel.

Control plane indirection. We assume that all packets are transmitted via $i3$. For most applications we expect the indirection overhead to be acceptable, but there might be applications for which achieving the highest possible throughput and lowest latency is critical. For those applications, one can implement a solution similar to TCP Migrate, where $i3$ is used only to exchange the new IP addresses when end-hosts move. In comparison to the basic TCP Migrate solution, such an approach would allow simultaneous mobility and would avoid overloading the DNS.

Home proxy. We assume that each end-host runs a ROAM proxy. In some cases, the robustness and efficiency this provides may not be worth the management and deployment costs. For example, during initial deployment, few of a MH's CHs will have ROAM proxies. An alternative is to deploy a home proxy for a MH that implements the functionality of the ROAM proxy for all of its non-ROAM CHs. This home proxy is analogous to the HA in MIP in that it is only used for hosts that cannot use a more efficient routing method.

Deployment issues. Our initial design assumes that ROAM uses a shared overlay infrastructure ($i3$). The most likely deployment strategy of such an infrastructure is still unclear. Options include a single provider for-profit service (like content distribution networks), a multi-provider for-profit service (like ISPs), and a cooperatively managed nonprofit infrastructure (like Gnutella). While full deployment is always hard to achieve, our solution is incrementally deployable; if the efficiency and robustness are not a concern, then it could start as a single server. Moreover, it does not require the cooperation of ISPs, so third parties can provide this service.

9 Conclusion

In this paper we present a highly robust and efficient mobility architecture. ROAM uses an indirection infrastructure ($i3$) that gives end-hosts the ability to *control* placement of indirection points in the infrastructure. ROAM uses this ability to achieve efficient routing, fast handoff, and preserve location privacy. ROAM is as robust as the underlying IP network, and allows simultaneous mobility while not requiring any changes to the TCP/IP protocol stack.

Simulation results show that ROAM has a low latency stretch and it is highly robust compared to Mobile IP. We evaluate a prototype of ROAM in a small testbed, and preliminary experimental results demonstrate that ROAM provides good support for soft-handoff and frequent mobility. We plan to deploy ROAM on a larger scale with end hosts and $i3$ servers spanning the continental US. In addition, we plan to explore using ROAM to compose services [6] such as transcoding and transport protocol optimization for wireless links (e.g., TCP Snoop [36]) that complement mobile routing.

References

[1] Charles Perkins, "RFC 3220: IP Mobility Support for IPv4," IETF, jan 2002.

[2] Alex C. Snoeren and Hari Balakrishnan, "An End-to-End Approach to Host Mobility," in *Proceedings of MobiCom*, 2000.

[3] N. Anerousis et. al., "TOPS: An Architecture for Telephony over Packet Networks," *IEEE JSAC in Comms*, 1999.

[4] Petros Maniatis et. al., "The Mobile People Architecture," in *ACM MC2R*, July 1999.

[5] Helen J. Wang et. al., "ICEBERG: An Internet-Core Network Architecture for Integrated Communications," *IEEE Personal Communications Magazine*, 2000.

[6] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana, "Internet Indirection Infrastructure," in *Proceedings of SIGCOMM 2002.*

[7] Mike Ritter et. al., "Mobile Connectivity Protocols and Throughput Measurements in the Ricochet MCDN System," in *Proceedings of MobiCom*, 2001.

[8] David B. Johnson and Charles Perkins, "Internet Draft: Mobility Support in IPv6," IETF, june 2002, work in progress.

[9] Stuart Cheshire and Mary Baker, "Internet Mobility 4x4," in *Proceedings of SIGCOMM*, 1996.

[10] Charlie Perkins and David B. Johnson, "Internet Draft: Route Optimization in Mobile IP," IETF, sept 2001.

[11] Pat Calhoun, Tony Johansson, and Charles Perkins, "Internet Draft: Diameter Mobile IPv4 Application," IETF, june 2002.

[12] Karim Malki et. al., "Internet Draft: Low Latency Handoffs in Mobile IPv4," IETF, june 2002, work in progress.

[13] Alper Yegin et. al, "Internet Draft: Fast Handovers for Mobile IPv6," IETF, march 2002, work in progress.

[14] Eva Gustafsson, Annika Jonsson, and Charles E. Perkins, "Internet Draft: Mobile IP Regionalized Tunnel Management," IETF, august 1999, work in progress.

[15] Hesham Soliman, Claude Castelluccia, Karim El-Malki, and Ludovic Bellier, "Internet Draft: Hierarchical MIPv6 Mobility Management," IETF, july 2002, work in progress.

[16] "Host Identity Protocol," `http://homebase.htt-consult.com/hip.html`.

[17] "LIN for IPv6," `http://www.lin6.net/`.

[18] Jerome Saltzer, "RFC 1498: On the Naming and Binding of Network Destinations," aug 1993.

[19] Faramak Vakil et. al., "Internet Draft: Supporting Mobility for TCP with SIP," IETF, december 2000, work in progress.

[20] Jayanth Mysore and Vaduvur Bharghavan, "A New Multicasting-Based Architecture for Internet Host Mobility," in *Proceedings of ACM/IEEE MobiCom*, 1997.

[21] J. Mysore and B. Vaduvur, "Performance of Transport Protocols Over a Multicasting-based Architecture for Internet Host Mobility," in *IEEE ICC*, 1998.

[22] S. Deering and D. Cheriton, "Multicast Routing in Datagram Internetworks and Extended LANs," *ACM TOCS*, May 1990.

[23] Ahmed Helmy, "A Multicast-based Protocol for IP Mobility Support," in *Proceedings of NGC*, 2000.

[24] Ahmed Helmy and Muhammad Jaseemuddin, "Efficient Micro-Mobility using Intra-domain Multicast-based Mechanisms (M&M)," Tech. Rep., USC, aug 2001.

[25] Frank Dabek, Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica, "Wide-area Cooperative Storage with CFS," in *Proc. ACM SOSP'01*, Banff, Canada, 2001, pp. 202–215.

[26] Diane Tang and Mary Baker, "Analysis of a Metropolitan-Area Wireless Network," in *Proceedings of MobiCom*, 1999.

[27] "Wireless LAN Medium Access Control and Physical Layer Specifications," Tech. Rep., IEEE Computer Society, 1999.

[28] Mark Corner and Brian Noble, "Zero-Interaction Authentication," in *Proceedings of MOBICOM 2002.*

[29] Mary G. Baker, Xinhua Zhao, Stuart Cheshire, and Jonathan Stone, "Supporting Mobility in MosquitoNet," in *Proceedings of USENIX Technical Conference*, 1996.

[30] R. Rivest, "RFC 1321: The MD5 Message-Digest Algorithm," IETF, apr 1992.

[31] Alex C. Snoeren et. al., "Hash-Based IP Traceback," in *Proceedings of SIGCOMM*, 2001.

[32] "GT-ITM," `http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html`.

[33] Shelley Zhuang, Kevin Lai, Ion Stoica, Randy Katz, and Scott Shenker, "Host Mobility Using an Internet Indirection Infrastructure," Tech. Rep., U.C. Berkeley, June 2002.

[34] "NIST Net Network Emulator," `http://snad.ncsl.nist.gov/itg/nistnet/index.html`.

[35] Jeremy Lilley, Jason Yang, Hari Balakrishnan, and Srinivasan Seshan, "A Unified Header Compression Framework for Low-Bandwidth Links," in *Proceedings of MobiCOM*, 2000.

[36] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz, "Improving TCP/IP Performance over Wireless Networks," in *Proceedings of MOBICOM 1995*.

Contact Networking: A Localized Mobility System

Casey Carter Robin Kravets
Department of Computer Science, UIUC
{ccarter,rhk}@uiuc.edu

Jean Tourrilhes
Hewlett Packard Labs
jt@hpl.hp.com

***Abstract*— MobileIP, the standard for Internet mobility, enables transparent mobility for a mobile node, but requires communication to take a multihop path through the node's Home Agent. Although a user with a multiple-interface mobile node may desire the ability to communicate locally, perhaps while disconnected from the Internet, MobileIP offers no such support.**

Contact Networking provides lightweight, localized network communication to a node with diverse network interfaces. The goal is to provide support for local connectivity equivalent to that provided by MobileIP for remote connectivity. The concept of link-layer awareness enables Contact Networking to tailor its operation to different links, using link-layer native services to implement abstract services when possible. Interface management and autoconfiguration insulate the user from concerns about the number and type of interfaces available.

In this paper, we motivate the need for localized mobility, and present the design and architecture of Contact Networking. Details of our prototype implementation illustrate the complexities of providing a localized mobility facility.

1. Introduction

Recent years have witnessed the development of a new class of mobile computing devices, merging personal communications and data access into a single device. One part mouse and two parts remote control, these mobile nodes are the interface point between users and the information environment. As users move through the world, they want to use their mobile nodes to maintain reachability through access to the global Internet, a form of mobility that is already well supported by MobileIP [21]. However, users would benefit from avoiding expensive multihop Internet connectivity and taking advantage of computing resources and services discovered locally, using whatever network interface is most convenient.

The inspiration for the name Contact Networking is to orchestrate direct communication between a mobile node and its neighbors. This focus on last-hop connectivity addresses the shortcomings of earlier work that assumes the presence of a network interface, with no concern for how that interface is link-layer configured. Contact Networking's approach to mobility support is diametrically opposite that of MobileIP. MobileIP treats all communication as though it is remote, even if local, whereas Contact Networking treats all communication as though it is local, even if remote. This difference in philosophy enables Contact Networking to provide lightweight service by being link-layer aware, tailoring itself to the characteristics of each link layer.

This work was sponsored by Hewlett Packard Labs and National Science Foundation grant number ANI–0081308.

Contact Networking manages both IP and link-layer configuration, connecting to neighbors using all available interfaces. Exploring the environment informs Contact Networking when communication is possible. As long as some pair of compatible interfaces exists, Contact Networking can maintain connectivity between neighbors. To preserve battery power and maximize flexibility, interface activation is delayed until applications indicate demand to communicate with a given neighbor. Contact Networking configures network interfaces to provide link-layer connectivity and bidirectional routing between the two nodes using a mechanism developed in earlier work [33]. Communication is maintained as links form and break by rerouting to other interfaces as necessary. This *neighbor handoff* mechanism, P-Handoff in earlier work [32], is transparent to the user and ensures persistent connectivity.

In addition to active interface configuration and routing support, Contact Networking provides name resolution to enable operation without infrastructure. The Contact Naming System (CNS) allows users to refer to local resources with friendly names. CNS is implemented by mechanisms native to each link layer. Since distant communication relies on the primitive hop-by-hop interaction of neighbors, Contact Networking extends to manage multihop end-to-end communications when infrastructure access is available.

In this paper, we present the design of and case for Contact Networking. Section 2 motivates better mobility support for mobile nodes, particularly localized mobility. We present the requirements for localized networking in Section 3, which is realized by the Contact Networking architecture of Section 4. Some examples detail the operational requirements for localized networking in Section 5. Section 6 discusses our prototype implementation of Contact Networking, including limitations and lessons learned. Section 7 contains the natural progression from Contact Networking to future work.

2. Motivation

We define local and remote communication, and detail the lack of support in mobility solutions for localized networking. We advocate link-layer awareness, the idea that the network layer should incorporate link-layer knowledge to provide better service. The following hypothetical scenario provides a context in which to discuss the issues of localized networking:

You and your colleague Xue board the train to travel to the client presentation. Xue hasn't seen the final version of the video, so you both point your PDAs' infrared ports at each other, and you push the video file to her PDA. Your PDAs connect over infrared, establishing a local communication channel without requiring Internet connectivity. Seconds after the transfer is initiated, you put your PDA back in your briefcase. This breaks the infrared link, so your PDA switches automatically to 5GHz radio to continue the transfer.

After a few moments, you decide to walk back to the dining car. Due to the limited range of 5GHz radio, that link also soon breaks, forcing the communication to fall back to 2.4GHz radio. As you walk away, you leave 2.4GHz radio range and your PDA's only option to preserve communication is to activate its GSM radio and communicate over the Internet with Xue's PDA. For the first time, your PDA registers with MobileIP and begins using multihop remote communication.

In the dining car, you use 5GHz radio to exchange business cards with a new acquaintance, without even needing to see his PDA – you simply push the card to whichever device in your vicinity responds to the nickname "Gerard." This transaction does not interfere with the ongoing transfer to Xue's PDA three train cars away.

As you return to Xue's car, the PDAs discover they can again connect to each other over local links. They shut down the expensive GSM radios and handoff the communication to the faster, cheaper 5GHz radio channel.

A. *Local and Remote Communication*

Mobile nodes carry out two fundamentally different kinds of communication, which we term local and remote communication. Local communication is between a mobile node and another node with which it shares direct connectivity. The defining characteristic is locality—interacting with "this" device that the user discovers in the environment, such as a local printer. A local communication is a simple unplanned association between two hosts, requiring no infrastructure access, but only that the two nodes share compatible communication hardware. Two users who meet each other and wish to exchange data between their mobile nodes, like the video in the scenario, use local communication.

Remote communication is between a node and some other explicitly addressed node that is not in the immediate neighborhood. Web browsing is the classic example of remote communication. The web browser knows the web server's identity, but not its location. The mobile node must have access to the network infrastructure to communicate. Unlike the case of local communication, there is no possibility of spontaneous interaction. In the scenario, when local communication links no longer provide connectivity, the PDAs automatically fall back to remote GSM communication over the Internet.

Remote communication is more resource-intensive than local. A short local transaction could be completed by briefly powering up a network interface and exchanging packets. A remote communication requires a mobile node to power up an interface, find an infrastructure provider, register with its home agent and then perform the communication before unregistering and powering the interface back down. The setup time necessary to register the mobile node's location, together with much longer delays incurred by the multihop path, keep the network interface powered over a longer time.

A mix of concurrent local and remote communication is not supported by current mobility approaches. MobileIP focuses on multihop remote communication [21]. MobileIP even explicitly forbid a node to directly communicate with any neighbor except its Foreign Agent:

> While the mobile node is away from home, it MUST NOT transmit any broadcast ARP Request or ARP Reply messages. Finally, while the mobile node is away from home, it MUST NOT reply to ARP Requests in which the target IP address is its own home address. . .

The PDAs in the scenario would be unable to communicate via MobileIP had access to the Internet been unavailable, even though they share compatible hardware. If a network administrator was available, the devices could be manually configured. The administrator must choose between two alternatives:

- Use the shortest range, lowest power link supported by both nodes. This conserves battery power for the mobile nodes, but constrains the users' motion within the limited range provided.
- Use the longest range, highest power link supported by both nodes. This provides a large area within which users can move freely. However, they pay for this flexibility with battery power.

Neither approach is optimal in all cases. From the users' perspective, the optimal solution would be to use the lowest power, shortest range links that provide connectivity at any given time. This dynamic approach is untenable for manual configuration.

B. *Multiple Wireless Technologies*

There is no perfect wireless technology for all applications. The design of a wireless technology is a tradeoff between range, capacity and power consumption. As soon as a technology comes to market it is obsoleted by newer, faster, more efficient technology. There will always be periods of transition from the current to the new wireless technology during which mobile nodes will support multiple links.

The management of multiple links is not integrated well into the traditional network stack. A multi-interface mobile node should be able to simultaneously utilize as many interfaces as necessary to satisfy its communication needs. In fact, users should not need to be aware of the number or type of interfaces available.

Among the many types of wireless technologies we pay special attention to a distinguished class we call *Directional Area Networking* technologies. These Directional links are characterized by short range and directional transmission characteristics, for example, infrared and laser. Although directional technologies are often deprecated, we see their limitations as features. Short range and directionality make directional technologies an ideal choice for a picking mechanism – selection of a particular device in the environment simply requires the user to point and click.

C. Link-layer Awareness

One of the best qualities of the Internet Protocol is its support for a variety of link layers by providing a uniform interface at the network layer. Although not every link layer was designed for IP, IP works over every link layer [3], [5], [10], [15], [23], [25], [26], [27], [29]. This least-common-denominator approach loses link-layer specific advantages. By restricting its expectations to simple datagram delivery, IP loses the richness of the individual link layers.

Link-layer awareness provides access to the capabilities of each link layer. The advantages are 1) lightweight discovery mechanisms for finding new neighbors, 2) tighter attachment to neighbor nodes, allowing rapid detection of link failure, and 3) the ability to perform link-specific optimizations, such as header compression, when appropriate. The richness of wireless links in particular makes link-layer awareness more attractive.

To clarify, consider two different wireless link layers: IEEE 802.11 [13], and IrDA (Infrared Data Association) [17]. Both links natively detect link breakage. These link failure mechanisms provide useful information to a link-layer aware network layer. The aware network layer can immediately stop using the broken route and more rapidly find a new route. In the absence of link-layer failure indications, some periodic messaging system must be used to monitor link integrity. MobileIP monitors links between mobile nodes and foreign agents with periodic beacons. Link-layer awareness conserves resources—the link layer monitors the link, whether the network layer pays attention or not—and can provide more rapid notifications than higher-layer monitoring. In general, link-layer awareness enables higher layers to transparently take advantage of implementations tailored to each link layer by providing abstract APIs for low-level services.

In the absence of link-layer awareness, the network layer must provide basic services like neighbor discovery. This duplication of service already in the link layer wastes resources. In addition, the network layer cannot possibly provide service of the same quality provided by the link layer mechanism without tailoring its approach to specific link layer characteristics. The end result of this approach is a fragile service only suitable on a single link layer, precluding true interface heterogeneity.

3. Design Requirements

The user experience of localized networking must be identical to that of infrastructure networking. The user should not need to use different interfaces or tools for local vs. remote communication. With that goal in mind, we propose the services necessary to support local connectivity and configuration-free multi-interface networking. We also address deployment, since a scheme to support localized networking is useless if not deployable in the current Internet.

A. Support Services

The critical services for localized networking are:

1) Neighbor Discovery
2) Name Resolution
3) On-demand Interface Binding
4) IP Autoconfiguration
5) Neighbor Routing
6) Channel Management
7) Infrastructure Access

We present a justification for each service requirement.

1) Neighbor Discovery: Neighbor discovery allows a mobile node to determine who its one-hop neighbors are. Neighbor knowledge enables a node to determine which interfaces may be used to contact a particular neighbor. In the absence of a neighbor discovery service, a mobile node would be incapable of performing local communication at all.

IPv6 Neighbor Discovery (IPv6ND) [18] is unsuitable for providing discovery to mobile nodes. First, its operation is not fully specified for multihomed hosts. Second, Neighbor Unreachability Detection may be too slow to support efficient handoffs when a neighbor is actively supporting a flow. Foremost, any IP-based solution obviously requires IP to be up and configured, conflicting with the goal of on-demand interface binding. Using IP as part of the mechanism to bootstrap IP creates a chicken-and-egg problem that is avoided by performing discovery at a lower layer.

2) Name Resolution: To support localized naming, a name-to-address mapping mechanism is necessary, like DNS provides in the Internet. If mobile nodes are to communicate directly each node must participate in the name-to-address mapping. Additionally, it is convenient for the user to select a nearby device without needing to know its proper name. For example, a name that denotes "the device my Directional Area Networking port is pointing at" enables the picking mechanism discussed in Section 2-B.

3) On-demand Interface Binding: Binding is the process of configuring an interface to use a particular link-layer medium, like plugging an Ethernet card into a jack. Interfaces have an unbound state in which the interface is not capable of full communication but may be able to perform some tasks, such as scanning for neighbors. Once bound, the interface can send and receive IP packets. Although they lack jacks, wireless interfaces also require configuration to select the link-layer medium. Even broadcast interfaces like 802.11 have parameters that bind the interface to one physical channel at a time. To be link-layer neighbors, two nodes must have interfaces that are bound to the same medium.

There are two reasons to delay interface binding. First, it is important to keep interfaces in a low-power state, since mobile nodes are battery powered. Second, some interfaces provide point-to-point links and can be used to communicate with only one neighbor at a time. To save power and allow flexibility in choosing to which link to bind, a local communication scheme must delay binding as long as possible. Therefore, interface binding should occur on-demand, when an application indicates demand to communicate with a neighbor. Certainly, it is inefficient to maintain connectivity with neighbors with which the node has no need to communicate. Interface unbinding must also be automatic when demand is removed.

4) IP Autoconfiguration: Interface binding does little good if the user must manually configure IP address and netmask. To enable transparent network access, interface IP configuration, including assigning IP address and netmask, must also be automatic.

Link-local (LL) addresses, in either IPv6 [6], [9] or IPv4 [4], provide autoconfiguration. Unfortunately, addresses with scope restricted to a single link provide little support for persistent communication. Transient addresses cannot be used as endpoints for a transport layer connection, or that connection will fail upon moving. There is no mapping from LL addresses to home addresses that identify mobile nodes. These two deficiencies render LL addresses little more than IP-layer aliases for link-layer addresses. Further, both IPv4 and IPv6 address require an initial period on the order of several seconds for duplicate address detection before the address can be used. This delay is hardly conducive to performing rapid handoffs across multiple interfaces.

5) Neighbor Routing: The final step necessary to neighbor communication is the establishment of symmetric routing state. Neighbor routes are all that is needed to complete the local communication. Once these routes are available, application layer communication can continue normally.

6) Channel Management: The previous services provide local communication to the mobile node, but do not shield the user from managing links and interfaces. Two abstractions are fundamental to the operation of channel management. First, a *connection* is a link-layer abstraction that is formed between two neighbors who wish to communicate. Presence of a connection indicates that a handshake has taken place between these neighbors signifying that both have agreed to the communication. Second, a *channel* is the end-to-end abstraction through which transport flows propagate. We say that the channel between two nodes is realized by a particular connection at any given time.

Although connections can break due to mobility, the goal of mobility support is to preserve the channel across these breaks. If the user is actively communicating with a neighbor through a channel over a connection that fails, the mobile node must maintain the channel by connecting over another interface if possible (see Figure 1). Similarly, if a mobile node is connected to a neighbor over technology A and a new, better path over technology B is discovered, the channel should automatically switch to the new interface.

7) Infrastructure Access: Enabling a mobile node to communicate locally is an achievement, but not if the local focus completely occludes remote communication. The ability to access the Internet is far too critical to sacrifice in the name of other goals. In addition to the locally-oriented scheme, remote communication is necessary to complete localized networking support.

B. Deployment Considerations

The goals of autoconfiguration, neighbor routing, and channel management are not achievable by only a single node of a neighbor pair. Bidirectional communication

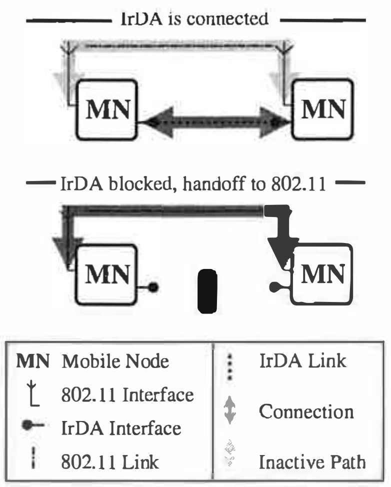


Fig. 1. Link-layer feedback and route failover

necessarily requires support from both neighbors. Consequently, it is impossible to provide localized communication support when directly communicating with legacy network nodes.

To be deployable, a localized communication scheme must not require support from every node in the Internet. The localized scheme can put no special requirements on applications; it should allow unmodified binaries to work with existing networking APIs. One way to achieve this level of transparency is to isolate the scheme at the network layer or below, so that transport and application layers see the standard IP network-layer interface. Applications and users are presented with the illusion of continuous network service, with no effect other than a temporary aberration in service during handoffs.

Application- and transport-layer mobility support advocates claim that applications must see mobility events in order to adapt to changed network service [20], [30]. Note that providing the illusion of network stability does not necessarily preclude the potential to provide mobility notifications to interested applications. It only reduces the impact on applications that are *not* mobility-aware.

4. Architecture

We present Contact Networking's architecture and discuss how its individual modules fit the requirements presented in Section 3. Since the requirements span route management to interface configuration, Contact Networking lies at the junction between the link and network layers in the OSI network stack, extending into both layers as depicted in Figure 2. Contact Networking includes a link-layer agnostic network layer module and several link-layer aware modules. The components of the network layer module manage routing and select between multiple paths available to a neighbor through different interfaces.

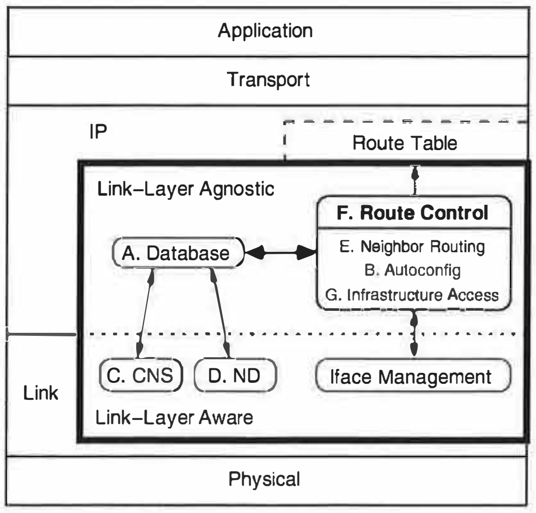


Fig. 2. Introducing Contact Networking into the OSI stack

Components of the link-layer-aware modules monitor the environment and report discovery events and link transitions. For local communication, link-layer modules also monitor link activity and inform the node when neighbors move out of range (i.e., a link breakage event). Contact Networking's link-layer discovery mechanisms incorporate the functionality of the Address Resolution Protocol (ARP) [24], mapping IP addresses to link-layer addresses.

Our description of Contact Networking architecture begins with its detailed component structure, as depicted in Figure 2. The specific architectural components discussed are:

A. the network database, where the mobile node stores its model of the environment,
B. the IP autoconfiguration technique, which assigns addresses to bound interfaces,
C. the Contact Naming System (CNS), a name resolution scheme for Contact Networking nodes,
D. the Contact Networking approach to neighbor discovery,
E. the neighbor routing component,
F. the routing control, which directs interface binding, neighbor routing, channel management and infrastructure access,
G. the infrastructure access component, which Contact Networking uses to access the Internet.

For each architectural component, we describe its responsibilities in fulfilling the design requirements, as well as how it interacts with other components.

A. Network Database

Contact Networking includes a database that models a node's current view of the environment. This database is the central communication point for all parts of the architecture. The four basic abstractions with which it is populated are interfaces, links, neighbors and paths.

- Interfaces are network attachment points on a node, such as Ethernet interfaces or cellular modems.
- Links are the communication media to which interfaces bind. An interface can bind to only one link at a time. Treating links as first-class entities makes it possible for the database to model both point-to-point links, with one neighbor per link, and broadcast links, with many neighbors per link.
- Neighbors are the nodes in the environment with which the user may wish to communicate.
- Paths are a pairing of a link and a neighbor augmented with the neighbor's address on that link.

An interface can bind to only one link at a time. Contact Networking can simultaneously establish connections with any and all neighbors to which it knows paths on a bound link. To establish communication with a neighbor, route control chooses a path to that neighbor

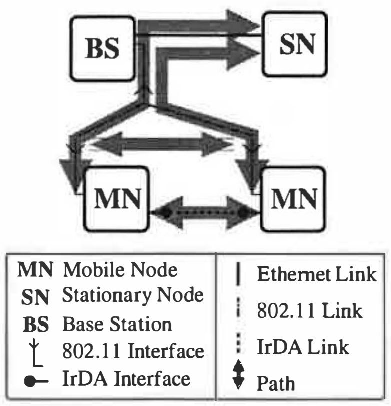


Fig. 3. Interfaces, Links, Paths, & Neighbors

from the database to activate. Path activation may necessitate interface binding and IP autoconfiguration using the configuration state for the link in the database. Contact Networking synchronizes path activation between neighbors, ensuring that both neighbors use symmetric routes. Once bidirectional path activate is complete, the paths are considered connected.

A path can be in one of four states:

- Available - The path is believed to exist but is currently inactive.
- Connected - The path is currently capable of transporting packets to the neighbor.
- Blocked - The path was connected, but is currently in a transient failure condition.
- Unblocked - The transient condition has passed, but the path has not been reactivated.

The blocked and unblocked states make it possible for Contact Networking to temporarily reroute traffic to another path while the primary path is recovering.

To clarify these abstractions, Figure 3 depicts a network composed of two mobile nodes and one stationary node. The base station simply bridges the wired and wireless links, and is therefore not visible to Contact Networking. The stationary node has one interface, and each of the mobile nodes has two. There are two links in the figure, one IrDA and one 802.11/Ethernet. The two bridged segments form a single link. Each mobile node knows a path to the stationary node, and two paths to its mobile counterpart, one over 802.11 and one over IrDA.

B. IP Autoconfiguration

A network interface must be configured with an address and subnet mask before IP can use it. In traditional IP., providing unique interface addresses is a problem in its own right, requiring either configuration or protocol solutions [4], [6], [7], [22]. These heavy-weight solutions require significant setup time to allocate an interface address, and create another problem when considering mobility—a transport flow cannot outlive the IP addresses used to identify its endpoints. If those addresses are transient, interface-specific addresses then mobility is impossible. The goal of mobility is to provide a communication channel that persists beyond the lifetime of a single address-to-interface configuration.

Contact Networking embraces and extends the MobileIP approach to providing a persistent communication channel in the face of link mobility. A *home address* is associated with the mobile node itself with no subnet mask. MobileIP configures the home address on a particular interface providing mobile networking through only that interface, or selectively configures the home address on the single interface that currently has connectivity, enabling vertical handoffs [31]. Contact Networking extends this notion to its natural conclusion, configuring the home address on *all* network interfaces simultaneously.

Due to the expanded role of the home address in Contact Networking, we designate it the Globally Routable IP address (GRIP). Every node is identified by a unique GRIP. Contact Networking requires GRIPs to be permanently assigned static home addresses, so that the presence and uniqueness of the GRIP can be guaranteed even when disconnected from the infrastructure. Contact Networking associates each neighbor's GRIP with its entry in the network database. Use of the GRIP for local communication in Contact Networking neatly sidesteps the problem of providing distinct addresses for each network interface. Configuring the GRIP on all interfaces allows the mobile node to persistently communicate through all interfaces simultaneously. The GRIP is guaranteed to live longer than any communication channel.

Using the same address on several interfaces is in direct conflict with the traditional IP addressing model in which one IP address denotes one network interface. Semantically, Contact Networking removes the emphasis that IP places on the role of the network interface. To Contact Networking, interfaces are transient entities, but the GRIP always stays the same. Since all application communication between nodes uses the GRIP, mobile nodes are accessible and successfully mobile as long as at least one network interface is present. The choice of which interface to use for a particular channel becomes a pure routing problem, a choice of path from the network database, and not a question of interface management or configuration.

C. Name Resolution

The Contact Naming System (CNS) is Contact Networking's name resolution and name-to-address mapping facility. CNS names are structured exactly like DNS names, with textual names separated by dots. CNS names enable users and applications to transparently control the operation of Contact Networking by specifying CNS names in place of DNS names.

Every node has a full CNS name, which is exactly the fully-qualified DNS name that resolves to its GRIP.

Agreement between CNS and DNS on the proper name for a node enables remote and local communication to use the same name.

CNS also supports link-layer specific name aliases, "wildcard" aliases and service aliases that give a user flexibility in choosing a neighbor. Link-layer specific aliases combine a CNS name prefix with a suffix denoting a particular link layer. This facility enables a user to specify that they want, for example, `bob.irda`, which resolves to any neighbor whose CNS name starts with `bob` reachable via IrDA. The wildcard alias `any` can be used to select any reachable neighbor, perhaps additionally qualified with a link-layer specific suffix. The infrared picking mechanism described in Section 2-B can be easily implemented with the alias `any.irda`.

A node replies to queries with its CNS record, containing its name and GRIP, and possibly service information. For example, the node `bob.cs.uiuc.edu`, which provides print service and acts as a MobileIP foreign agent, might have the following CNS record:

GRIP: `128.174.244.37`
Name: `bob.cs.uiuc.edu`
Services: printer, MIPFA

Contact Networking attaches each CNS record to the corresponding neighbor record in the network database. Name queries that can be satisfied from the database require no network traffic. Link-specific alias queries that cannot be answered from the database only result in network traffic on links of the specified type. An request to resolve `bob.irda` queries for `bob` only on IrDA.

CNS is similar to other multicast name resolution schemes [1], [8], [19]. The main difference is in implementation. Our goal of link-layer awareness means that CNS is transported by link-layer specific mechanisms instead of relying on IP multicasting support as in other approaches. Furthermore, the lifetime of CNS records is tightly coupled with neighbor knowledge in the network database. This tight coupling allows CNS records to be cached long-term without refreshing, as long as the neighbor is still present.

Each node monitors its interfaces for CNS queries requesting its configuration, although CNS may not use this request-reply model on every link layer. A Contact Networking node uses link-layer native mechanisms when available to disseminate and retrieve CNS records. Some links perform discovery while in the unbound state, so link-layer aware CNS conserves power and lowers delay over an approach that uses IP to resolve names. Nodes with sufficient power may also actively advertise their CNS records either through a link-layer discovery method or by periodic broadcast. These advertisements enable other interested nodes to easily collect CNS records by passive listening.

A node returns its CNS record in response to a the following types of CNS queries:

- Name queries
- GRIP queries, enabling reverse lookup
- Service queries (e.g., a node that provides MobileIP Foreign Agent service might respond to the name `MIPFA`)
- Wildcard queries, possibly link-layer qualified

These CNS query types provide several ways for users to choose machines in their environment with which to interact, without requiring changes to IP network software that uses DNS names. CNS access is integrated alongside the normal DNS name resolution mechanism on the nodes, so that application name lookups try CNS resolution before DNS. A unified name resolution API makes localized networking easily available to network applications. For example, to purchase a can of Dr. Pepper^{TM} from a vending machine, a user would point the infrared port of their device at the vending machine and point their web browser at `http://any.irda`. The CNS resolver would return the vending machine's CNS record that specifies its name and GRIP. An exchange of business cards with another user named Bob would probably use the name `bob` to locate the GRIP of Bob's mobile node. We call this "getting a GRIP" on the destination.

D. Neighbor Discovery

Neighbor discovery must incorporate link-layer awareness. Determining when another node is within the sphere of communication of a particular interface on a particular link is necessarily link-layer specific. Link-layer adaptation modules perform neighbor discovery and indicate neighbor appearance/disappearance to the link-layer agnostic module. Each link-layer aware discovery mechanism is tuned to the particular link, using native discovery mechanisms when available. This level of link-layer awareness provides high quality discovery on each link layer without compromising the generality of the rest of the Contact Networking system.

What other systems term neighbor discovery, finding a reachable peer on a link, is really path discovery in Contact Networking. Neighbor discovery in Contact Networking is only complete after two neighbors have exchanged CNS records. After the exchange of CNS records Contact Networking updates the network database with a new path entry and a new neighbor entry, if the neighbor is not already present.

Contact Networking's use of CNS records for neighbor discovery combines neighbor discovery, address resolution and name resolution into a single protocol. This combination reduces total overhead in contacting a neighbor, as well as allowing for long-term caching of name-to-IP address and IP-to-LL address mappings. The tight coupling of address mappings to neighbor lifetime means that mappings need never be refreshed and are simply flushed when a neighbor goes away.

E. Neighbor Routing

Contact Networking takes link-layer network semantics, a flat address space with one-hop communication, and presents it to the transport layer as a set of host-specific routes. This IP facade on link-local connectivity enables IP applications to use the localized networking facility equivalently to traditional IP support.

Routing necessarily affects the address model of Contact Networking. The use of GRIPs for local communication creates a class of *local* routes administered by Contact Networking. There is exactly one such route corresponding to each neighbor in the network database. A local route can be inactive, serving as an indication that Contact Networking can provide a path to that neighbor on demand, or active, enabling packet forwarding across some connected path.

Traditional IP routing, which views IP addresses as specifying a location within the internetwork, is retained in Contact Networking for routing to remote destinations. The result is a bipartite routing system that provides maximum flexibility in routing directly to neighbors—via local routes–but does not lose the ability to interface to the Internet with traditional routes. Integrating both styles of routing enables transitional deployment. Contact Networking mobile nodes can contact the current Internet through access points much like MobileIP foreign agents that provide Contact Networking service in addition to traditional IP routing. In the absence of Contact Networking-enabled access routers, a mobile node must resort to traditional techniques for unreachability detection and cannot provide rapid handoff when the link to the access router fails.

F. Route Control

Route control in Contact Networking handles on-demand interface binding and channel management. When a Contact Networking node first wishes to contact some IP address, it performs a lookup in the network database for a neighbor whose GRIP is that address. If the neighbor entry is present, its CNS record and at least one path will be known—the path over which the neighbor was discovered. Route control selects one of the available paths to connect and negotiates connection with the neighbor. If no record is present, the node falls back on traditional IP networking to find a remote route to the destination, possibly initiating infrastructure access as detailed in the next section. Route control also performs channel management by disconnecting idle connections and unbinding idle interfaces when the last connection on that interface is dropped. When better paths are discovered, route control actively switches channels to better connections.

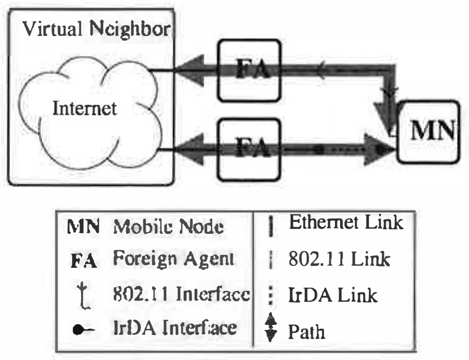


Fig. 4. Infrastructure as a virtual neighbor

G. Infrastructure Access

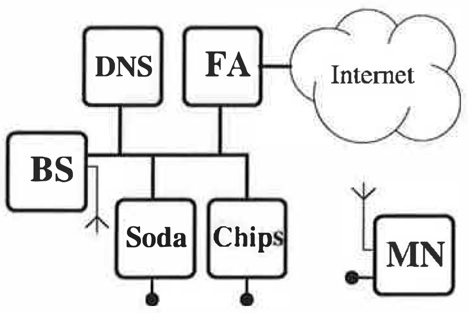
Since Contact Networking views all communication as direct neighbor communication, some extension is necessary to support remote communication. Contact Networking treats remote communication as local communication with a virtual neighbor that represents the entire infrastructure network. This conversion of remote to local communication enables Contact Networking to apply its normal mechanisms for on-demand establishment and channel management to infrastructure access.

In Contact Networking, a network node that wishes to provide infrastructure network access to others by performing as a MobileIP Foreign Agent advertises its willingness to do so through the Contact Naming System. Mobile nodes that need infrastructure access then discover the infrastructure provider normally through neighbor discovery. Upon discovering an access provider, Contact Networking creates a virtual path to the virtual neighbor that corresponds to the real path to the access provider (see Figure 4).

Integrating dial-up access into this architecture is trivial. A dial-up phone number is essentially a path description for a type of virtual path. Whenever a mobile node can use a dial-up interface, it is one hop away from the virtual neighbor. This virtual path would be statically configured into Contact Networking, with a simple monitor module that "discovers" the virtual path when a modem is plugged in or when within GSM coverage.

Contact Networking route lookups for IP addresses which do not correspond to neighbors in the network database are redirected to the virtual neighbor. Connecting a path to the virtual neighbor also connects its corresponding real path and performs MobileIP registration through the neighbor on the real path.

If a mobile node needs to maintain Internet access to enable externally initiated channels, Contact Networking can be configured to never drop virtual connections due to idleness. Normal neighbor handoff mechanisms would then preserve the channel to the Internet whenever possible, keeping the mobile node accessible to the rest of the world.



MN	Mobile Node	**Chips**	Vending Machine
BS	Base Station	Ỵ	802.11 Interface
DNS	DNS Server		
FA	Foreign Agent	●–	IrDA Interface
Soda	Soda Machine		

Fig. 5. Example network from Section 5

5. Examples

Some simple examples serve to illustrate the operation of Contact Networking. For these examples, we discuss a hypothetical user, Prashant, with a mobile node that has infrared, wireless LAN and wired Ethernet interfaces. Prashant is a Computer Science graduate student, who happens to be wandering around the CS building. The GRIP of Prashant's mobile node is topologically located on the Ethernet LAN. There is a building-wide wireless LAN, which has an attached DNS server and a MobileIP Foreign Agent, as in Figure 5. All machines on the CS building network are Contact Networking enabled.

1) Simple Local Communication: In the first scenario, Prashant is walking through the Computer Science building, when he passes a vending machine. Feeling a grumbling in his belly, Prashant decides to purchase a bag of Fritos^{TM}. We hypothesize the existence of software in Prashant's mobile node and the vending machine that is capable of automatically performing a small cash transaction. To initiate the transaction, Prashant points his device at the vending machine's infrared port and clicks the "pay" button. A flurry of events ensues.

The application on Prashant's machine tries to resolve the CNS alias `any.irda`. The IrDA-specific CNS module performs native IrDA discovery to detect the vending machine nearby. As a part of the discovery process, Prashant's node and the vending machine exchange CNS records containing their GRIPs and DNS names. As a side effect of CNS discovery, Contact Networking routes are created to enable on-demand connection of the two machines. The CNS resolver on Prashant's machine returns a `struct hostent` to the application, informing it that `any.irda` is in fact an alias for `chips.cs.uiuc.edu` with GRIP `128.174.244.91`.

Having obtained an IP address for the desired neighbor, Prashant's application initiates a TCP flow to the application in the vending machine. The first packet of this flow is queued while the path to the vending machine is connected. Since no better path to the vending machine is known, Contact Networking connects the IrDA path. A full path connection occurs:

1) The IrDA interface on the mobile node is bound to the link between it and the vending machine.
2) The mobile node establishes a connection to Contact Networking on the vending machine.
3) Contact Networking on the vending machine accepts the connection request and both nodes create local routes indicating that the channel between the two GRIPs is using this particular connection.

Upon establishing the connection Contact Networking forwards the queued data packets to the vending machine. Transaction processing then continues normally with no further involvement from Contact Networking. Prashant grabs his bag of Fritos^{TM} and starts walking back to his office to write a paper about scatternet formation in Bluetooth. Within several seconds, the IrDA module on each machine informs Contact Networking that the connection has become idle. Contact Networking drops the idle connection and unbinds the IrDA interface since no other connections exist on the link.

2) Local Communication with Handoff: On his way to the office, Prashant passes a soda machine and decides to purchase a can of Coke^{TM} to help wash down those salty Fritos^{TM}. The transaction proceeds similarly, until the impatient Prashant returns his mobile node to his pocket, interrupting the TCP flow over IrDA. The IrDA module in Prashant's mobile node informs Contact Networking of the premature connection breakage. Since the connection was being actively used by data traffic, Contact Networking immediately tries to find an alternate path to the soda machine in the network database.

Luckily for Prashant, the soda machine is also connected to an Ethernet segment bridged to the Wireless LAN. Contact Networking connects this alternate path. During the handoff procedure, no connection is available to support the channel to the soda machine, so any data packets are queued. After the neighbor handoff completes successfully, the channel is rerouted over the new connection and the queued packets are retransmitted. TCP on Prashant's node retransmits the single lost packet and recovers the transaction without the application's or the transport layer's awareness or participation. As in the previous example, the transaction completes normally and the connection becomes idle. Contact Networking at both ends drops the connection.

3) Infrastructure Access: On the way back to the office, Prashant decides to load Slashdot into Mozilla to check for interesting news—which can be challenging, with a bag of Fritos^{TM} in one hand and a can of Coke^{TM} in the other. CNS resolution of `www.slashdot.org` fails since Prashant is nowhere near the Slashdot web server. When his DNS resolver tries to resolve `www.slashdot.org`, Contact Networking connects the path to the DNS server to complete the resolution, which returns an IP addresses for Slashdot's web server.

In this case, the first TCP packet of the flow is *not* destined for a neighbor in the network database, so Contact Networking connects its virtual path to the Internet:

1) Virtual path connection activates the underlying path to the Foreign Agent on the wireless LAN.
2) Once the Contact Networking connection to the Foreign Agent is established, MobileIP registration takes place.
3) When registration is complete, the virtual path is considered to be connected.

The TCP packet is finally forwarded over the newly established channel to Slashdot and Prashant's browser retrieves the web page normally. Several seconds after the page load completes, the virtual path becomes idle, forcing MobileIP deregistration and disconnection from the Foreign Agent.

6. Implementation

The prototype implementation of Contact Networking is developed on Linux atop a modified kernel. The implementation supports three different link types: infrared (IrDA), Ethernet and 802.11. The centerpiece of the implementation is the Connectivity Manager, a userspace daemon that runs on each Contact Networking node. Transparent application access to CNS is realized by an extension for the name service switch library.

A. *Kernel Modifications*

We extended the Linux 2.5 kernel by providing a mechanism for wireless interface drivers to export events to userspace programs. This extension became Wireless Extensions version 14 and has since been integrated into the development series Linux kernel at version 2.5.7, as well as back-ported into the stable kernel series at version 2.4.20. One event type we included in the Wireless Extensions is link-layer delivery failure notification. The drivers for Orinoco 802.11 cards (`orinoco_cs`), Aironet cards (`airo_cs`) and cards based on the Intersil PrismII chipset (`hostap_cs`) were modified to send these notification events. These driver modifications are in various stages of integration into the kernel sources.

B. *Connectivity Manager*

In Contact Networking, each computer hosts a Connectivity Manager (CM) that configures its network interfaces and coordinates its discovery of and connection to neighbors, as well as maintaining the Contact Networking network database. The CM provides a common rendezvous point between user programs, the network stack and the routing tables.

1) Management and Adaptation Layers: The CM is organized as a process that reacts to asynchronous events. Low-level events are passed up from the link-layer specific adaptation modules, collectively denoted the adaptation layer, to the link-layer agnostic management layer. The management layer responds to events by directing the adaptation modules to connect or disconnect paths, or by modifying the network database. The management layer also responds to application name resolution queries from the Name Service Switch (NSS) interface (see Section 6-C), searching the network database for appropriate CNS records.

Adaptation modules signal the management layer when the following events occur:

- An interface is inserted/removed from the node
- A link is discovered/lost
- A neighbor is discovered/lost
- A path to some neighbor is discovered/lost
- A path has changed state
- A connection request was received from a neighbor

In response to these signals, the management layer uses an abstract API to manipulate interfaces, links, and paths. Each adaptation module provides a concrete implementation of the abstract API with behaviors appropriate to its link technology. The management layer uses these APIs to direct an adaptation module to bind/unbind an interface to a link, connect/disconnect a path on a bound link, or reject/accept a connection request from a neighbor by connecting the return path.

Contact Networking takes an approach different from that of traditional IP in providing network layer services. Contact Networking is aware of the characteristics of each link layer and uses link-layer specific functionality to provide better network service. Contact Networking has a link-layer specific adaptation module for each supported link type. Adaptation modules implement CNS querying and neighbor discovery in a lightweight manner, using native link-layer discovery mechanisms when available. Link-layer modules also inform the CM of dynamic interface insertion and removal or conditions that represent failure of a connection.

Link-layer modules also provide mechanisms to form connections to neighbors – path connection sets up bidirectional connectivity with the neighbor over the chosen path, binding an available interface to the path's link if necessary. Once a path is connected, the link-layer module monitors the connection for breakage with link-native mechanisms (e.g., failure to receive an acknowledgment on 802.11). Link-layer connection break indications cause the CM to reroute the channel to that neighbor over another path, if possible. Removal of a bound interface also causes rerouting to occur for any channels using that interface. The adaptation modules monitor connections for idleness and signal the management layer to drop the connection. Disconnection tears down the neighbor connection, causing the interface to unbind from the link upon disconnecting the last connection. Unbinding and powering down the interface at the earliest possible time enables power conservation and allows the interface to be used to bind to other links.

All link-layer adaptation modules use a Linux `rtnetlink` socket to monitor interface creation events [28]. This enables each adaptation module to report interface discovery/loss events to the management layer.

The interface management aspect of Contact Networking is similar to Physical Media Independence (PMI), "an architecture for dynamically diverse network interface management" [14]. PMI observes the availability of network interfaces and informs applications of changes in interface state and availability, as well as coordinating a vertical handoff of the default route. Unlike the active interface management of Contact Networking, PMI passively reacts to network interface state changes.

2) IrDA Adaptation Module: The IrDA [17] adaptation module uses mechanisms native to IrDA and IrNET to provide services to the management layer. IrDA has a native discovery mechanism and a link-layer attribute exchange protocol (IrIAP). Path connection is realized by IrNET, which uses synchronous PPP for control, operating directly on IrDA frames. IrNET generates events to signal the IrDA adaptation module.

Upon receiving an IrNET discovery event, the IrDA adaptation module uses IrIAP to exchange CNS records with the new neighbor and delivers the resulting CNS record along with a neighbor discovery signal to the management layer. Since IrDA is a point-to-point technology, paths and links are unified in the IrDA adaptation layer. The architectural constraint of binding an interface to at most one link at a time mirrors the point-to-point nature of the link layer in Contact Networking. When the corresponding IrDA neighbor un-discovery event is reported from IrNET, the adaptation module reports both a lost path and a lost link event to the management layer.

Link block and connection request events are passed unmodified up to the management layer. The adaption layer maps link-layer identifiers into the database entry for the path corresponding to the blocked connection or the requesting neighbor.

The IrDA adaptation module, when instructed to connect a path by the management layer, forks a child `pppd` on the IrNET control channel. The `pppd` "connect script" argument is actually directions for IrNET to choose which neighbor to connect. The binding completes when the IrNET event channel reports successful connection, which is signaled to the management layer. If PPP/IrNET connection fails, the `pppd` will exit prematurely and the IrDA adaptation module notifies connection failure to the management layer.

To force disconnection, the IrDA adaptation module simply kills the child `pppd`. When the connection has finally shut down, IrNET will report disconnection on its event channel, at which time the IrDA adaptation module marks the interface in the network database as once again available for binding to other links and reports disconnection to the management layer.

Forking a child `pppd` process for neighbor connection is a heavyweight procedure, so using IrIAP to exchange CNS records below IP provides low latency discovery, avoiding IP setup altogether. On a connection-oriented link layer like IrDA or Bluetooth, it is unclear how one would perform network-layer discovery. To which neighbor do you connect your link layer before sending, say, an ARP request? Contact Networking's link-layer awareness enables support for discovery on connection-oriented link layers.

3) Ethernet/802.11 Adaptation Module: 802.11 [13], besides being wireless, is as different from IrDA as possible. 802.11 provides a simple datagram-delivery-port abstraction, with no additional frills built in. These facilities must then be directly coded in the 802.11 adaptation module. Interestingly, the additional code this creates is traded for code the IrDA module uses to monitor the external `pppd`. Both modules weigh in at almost exactly 2000 lines of C.

The 802.11 layer requires protocol support to handle many issues that IrNET/IrDA does not. Consequently, we devised a set of protocol commands that are encodable in a trivial subset of XML to facilitate easy communication between neighbor adaptation modules. This message encoding is used for CNS transport, path connection, path disconnection and idleness monitoring.

Lacking IrIAP, the 802.11 adaption module must provide an alternate means to exchange CNS records. A UDP broadcast request/reply protocol enables the adaptation layer to query the CNS records of its neighbors on a link or reply to their CNS queries. Since CNS record exchange must precede any connection between two neighbors on the link, a CNS request necessarily contains an advertisement for the source. Neighbors use the CNS request/reply as an opportunity to record MAC addresses and GRIPs. The adaptation layer creates an ARP table entry for the neighbor's GRIP with the discovered MAC address. The ARP entry's lifetime is coupled to the CNS record, avoiding ARP resolution altogether for a neighbor whose MAC address is already known.

Neighbor CNS records discovered over 802.11 are immediately reported to the management layer. Unlike with IrDA, it is possible to discover many paths to different neighbors on the same 802.11 link. Path connection for 802.11 requires only a simple handshake of XML commands to ensure bidirectional connectivity, so connection is much lighter weight and lower latency than IrDA.

The 802.11 adaptation module monitors an `rtnetlink` socket for Wireless Event reports, specifically packet delivery failure notifications from the device driver. The device driver reports the unreachable neighbor's MAC address, which is used to locate the path corresponding to that neighbor in the network database. Currently, the prototype treats a single failure indication as path blockage and reports it as such to the management layer. Future work will analyze this decision to determine if some level of hysteresis may provide better performance than single-packet-loss path blockage.

Although connection establishment needs only a simple request/reply and a single route creation, connection monitoring for 802.11 is greatly complicated over IrDA. In IrDA, monitoring the interface's send/receive packet counters is sufficient to indicate idleness of link and path. In 802.11, it is possible to connect many paths over the same interface, so that it is impossible to determine *which* path or paths are idle by simply examining the packet send/receive counters. Our prototype uses the Linux iptables firewall support to add an iptables rule for each connection that only recognizes traffic on that connection. The adaptation layer can determine connection idleness by reading the iptables rule's packet counter.

The iptables rules created by the 802.11 adaptation layer are a rather problematic piece of state in that they can outlive the CM process. While under development and prone to crashing, the CM tends to litter the iptables with stale rules. To avoid this problem, the adaptation layer forks a child process that clears out all associated iptables rules when/if the CM exits unexpectedly.

Path disconnection is relatively simple. At the management layer's behest, or in response to a neighbor's disconnect request, the 802.11 adaptation layer destroys the corresponding route and iptables rule and sends a disconnect command to the neighbor. After the neighbor confirms disconnection, the final path disconnect notification can be reported to the management layer.

A future goal for the 802.11 adaptation module is to support access point scanning to determine the Extended Service Set Identifiers (ESSIDs) of nearby access points. Each distinct ESSID discovered can be modeled as a distinct link in the network database. Although we added access point scanning support to Wireless Extensions v14, time did not allow completely integrating this feature into the prototype.

4) Virtual Adaptation Module: This adaptation module exists to support infrastructure access via virtual paths to the virtual neighbor. To Contact Networking, the single virtual neighbor represents the entire Internet. In Figure 4, the mobile node has two paths available to the infrastructure through different providers. When informed of the discovery of a path P to a neighbor N that advertises infrastructure service, the management layer directs the virtual adaptation module to create a corresponding virtual path P' with the virtual neighbor as its destination. The path P' is dependent on P, so that destruction of P causes destruction of P'. Semantically, this path dependency asserts that infrastructure access on the path P' is only possible while P is reachable. The dependency relationship between P' and P implicitly captures the fact that the node must direct packets destined for remote hosts through its neighbor N.

Requests from applications to contact non-neighbors are treated as requests to contact the virtual neighbor. When the user requires remote access, the normal path selection mechanisms in the management layer choose a virtual path P'. Connection of P' causes connection of the underlying real path P and initiates MobileIP registration through the neighbor N that P reaches, as well as creation of a default route with next hop N.

Virtual paths are monitored just like other paths. Therefore, vertical handoff for remote communication naturally falls out as an effect of neighbor handoff between virtual paths. When applications communicate with remote hosts, Contact Networking sees traffic over the virtual connection to the virtual neighbor. A break of the underlying connection is treated as a break of the virtual connection, which will cause the management layer to reroute the infrastructure channel over an alternate virtual path to some other infrastructure access provider.

5) Packet Handling: Contact Networking uses on-demand path connection and interface binding to provide maximum flexibility with minimum resources. To provide on-demand path connection, the CM requires a mechanism to intercept packets from local applications for neighbors to which it has valid paths. We call this mechanism the *demand channel*. Additionally, the demand channel must also be capable of capturing *any* non-local packets to support the virtual neighbor. Captured packets are queued until a path to the destination can be connected and then forwarded over the fresh connection.

The demand channel is implemented as a set of routes in the kernel routing table, one per neighbor, that points into a universal tunnel interface. The Linux universal tunnel is a device driver that attaches a network interface to a device file. Packets routed through the interface by the kernel may be read from the file and packets written to the file appear to have been received on the network interface. Packets thus captured are buffered in user-space.

Contact Networking also needs a mechanism to resend packets that it buffers during interface binding and path connection. Simply writing the packets into the universal tunnel will not suffice, since the mobile node is probably not configured to support packet forwarding. Our CM prototype uses a raw IP socket to inject unmodified captured packets into the network stack.

C. Application CNS Interface

The final piece to complete the Contact Networking architecture is a transparent application interface to CNS. Name resolution in Linux is handled through the Name Service Switch (NSS) library. NSS dispatches name/address queries to multiple name services, notably DNS and NIS. Contact Networking inserts a CNS name resolver into the NSS configuration, so that application queries try CNS before other naming services. Application queries are coded in XML and sent to the CM.

The name mapping component of the CM first tries to resolve queries from the network database. Failing that, any action taken to resolve the name is dependent on its structure. If the name ends in a suffix that identifies it as a link-layer specific alias—Currently `irda` for IrDA and `wlan` for 802.11—the name is passed to the appropriate adaptation module for on-demand resolution. The link agnostic suffix `adhoc` specifies that all adaptation modules should try to resolve the name on their links. Otherwise, the name mapping component returns a failure indication to NSS, which then continues to try to resolve the name with other name services as normal.

The wildcard alias `any` is handled by matching some appropriate neighbor entry in the network database. It is possible for naming conflicts to occur with wildcard aliases. For example, `any.irda` probably resolves to a single neighbor, but `any.wlan` or `any.adhoc` potentially aliases a large number of neighbors. When name ambiguity occurs, the name mapping component currently reports a "name not unique" error back to the original application. A more powerful mechanism for name disambiguation, possibly with user assistance, is a topic for future work.

7. Conclusions and Future Work

The main contribution of Contact Networking is to provide a complete solution for communication management on a local scale. Support for naming, routing, channel management and interface configuration in Contact Networking place localized networking on par with traditional Internet mobility. Contact Networking provides everything necessary for neighbor communication in the absence of the usual infrastructure service, while preserving ease of integration with the Internet when available. Unlike earlier work, Contact Networking is not dependent on the presence of the Internet for operation.

Contact Networking realizes the goals of a mobile node with multiple wireless interfaces through the application of link-layer awareness. Support for local communication differentiates Contact Networking from traditional mobility solutions that are infrastructure dependent. The Contact Naming Service provides on-demand naming and basic service discovery using link-layer native support. There are some important avenues we intend to explore to enhance Contact Networking.

Flexible Network Support for Mobile Hosts [35] extends MobileIP with support for multiple packet delivery methods, including MobileIP with or without reverse tunneling and/or route optimization, and regular IP. A mobile policy table enables selective mobility on a per-flows basis. Further, Flexible extends the MobileIP home agent to allow a mobile node to extend this flow-granularity routing support back to the home agent. The Flexible home agent enables different traffic flows to be routed to different care-of-addresses of the mobile node. In this way, Flexible enables mobile support for multiple interface remote communication.

Integration of Contact Networking and Flexible into a single system, along with a policy control to direct which traffic should use which interface, is an interesting avenue of future research. Allowing users to express preferences for network service to the CM would greatly increase the usability and applicability of the system.

Currently we are improving the interface management mechanisms present in Contact Networking. The approach we call co-link configuration will allow two devices to negotiate common link-layer binding configuration for shared interfaces. Co-link uses one interface to bootstrap others. For example, two PDAs could use IrDA to exchange a frequency and encoding key for secure 802.11 communication.

With no authentication, the local discovery and route management components of Contact Networking enable a local attacker to masquerade as an Internet host of the attacker's choice. This vulnerability is a consequence of the absence of infrastructure support and can be seen as a magnification of the ARP security hole [2]. Peer-to-peer authentication approaches from ad hoc networking could address this issue in Contact Networking [12].

Other work has suggested transport protocols that can aggregate bandwidth from multiple interfaces simultaneously [11], [16]. Contact Networking provides a framework for discovery of multiple paths that these protocols can use. We are investigating the integration of Contact Networking with an aggregating transport protocol.

Many network services have static or infrastructure-dependent configuration. Notable among these services is DNS. A node is usually configured statically with a set of DNS servers. DNS, and similar services, could possibly benefit from the local service discovery model that CNS makes available. A mobile node that wishes to avoid the expense of remote communication could then use local DNS servers that it discovers through CNS. Generalization of this approach to other services with similar requirements would also be useful. Use of an IP-based protocol for service discovery, such as SLP [34], necessitates IP configuration before service discovery may take place. An IP-based protocol cannot, therefore, be used to bootstrap IP.

Availability

We regret that we are unable to make our prototype available at this time. The current lack of documentation makes it nearly impossible for outside interests to install the prototype. We do intend to make Contact Networking available soon; interested parties should contact the first author at `ccarter@uiuc.edu`.

References

[1] *AppleTalk Network System Overview*. Addison-Wesley Publishing Company, Inc., 1990.

[2] S. Bellovin. Security problems in the TCP/IP protocol suite. *ACM Computer Communications Review*, 19(2), April 1989.

[3] C. Brown and A. Malis. Multiprotocol interconnect over frame relay. Request for Comments (Standard) RFC 2427, Internet Engineering Task Force, September 1998.

[4] S. Cheshire, B. Aboba, and E. Guttman. Dynamic configuration of IPv4 link-local addresses. Internet Draft (Work in Progress) `draft-ietf-zeroconf-ipv4-linklocal-07.txt`, Internet Engineering Task Force, August 2002.

[5] R. Cole, D. Shur, and C. Villamizar. IP over ATM: A framework document. Request for Comments (Informational) RFC 1932, Internet Engineering Task Force, April 1996.

[6] A. Conta and S. Deering. IPv6 stateless address autoconfiguration. Request for Comments (Draft Standard) RFC 2462, Internet Engineering Task Force, December 1998.

[7] R. Droms. Dynamic host configuration protocol. Request for Comments (Draft Standard) RFC 2131, Internet Engineering Task Force, March 1997.

[8] L. Esibov, B. Aboba, and D. Thaler. Linklocal multicast name resolution (LLMNR). Internet Draft (Work in Progress) `draft-ietf-dnsext-mdns-12.txt`, Internet Engineering Task Force, August 2002.

[9] R. Hinden and S. Deering. Ip version 6 addressing architecture. Request for Comments (Standards Track) RFC 2373, Internet Engineering Task Force, July 1998.

[10] C. Hornig. Standard for the transmission of IP datagrams over Ethernet networks. Request for Comments (Standard) RFC 894, Internet Engineering Task Force, April 1984.

[11] H.-Y. Hsieh and R. Sivakumar. A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts. In *ACM Mobicom '02*, pages 83–94, 2002.

[12] J.-P. Hubaux, L. Buttyán, and S. Čapkun. The quest for security in mobile ad hoc networks. In *Proc. of the ACM Symposium on Mobile Ad Hoc Networking & Computing (MobiHOC '01)*, pages 146–155, October 2001.

[13] IEEE Computer Society. 802.11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications, June 1997.

[14] J. Inouye, J. Binkley, and J. Walpole. Dynamic network reconfiguration support for mobile computers. In *Proceedings of the Third Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '97)*, pages 13–22, September 1997.

[15] D. Katz. Transmission of IP and ARP over FDDI networks. Request for Comments (Standard) RFC 1390, Internet Engineering Task Force, January 1993.

[16] L. Magalhães and R. Kravets. Transport level mechanisms for bandwidth aggregation on mobile hosts. In *9th International Conference on Network Protocols ICNP 2001*, 2001.

[17] P. J. Megowan, D. W. Susak, and C. D. Knutson. IrDA infrared communications: An overview. http://www.irda.org.

[18] T. Narten, E. Nordmark, and W. Simpson. Neighbor discovery for IP version 6 (IPv6). Request for Comments (Standards Track) RFC 2461, Internet Engineering Task Force, December 1998.

[19] NetBIOS Working Group. Protocol standard for a NetBIOS service on a TCP/UDP transport: Concepts and methods. Request for Comments (Standard) RFC 1001, Internet Engineering Task Force, March 1987.

[20] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and R. W. Kevin. Agile application-aware adaptation for mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP) '97*, 1997.

[21] C. Perkins. IP mobility support for IPv4. Request for Comments (Proposed Standard) RFC 3344, Internet Engineering Task Force, August 2002.

[22] C. E. Perkins, J. T. Malinen, R. Wakikawa, and E. M. Belding-Royer. IP address autoconfiguration for ad hoc networks. Internet Draft (Work in Progress) `draft-perkins-manet-autoconf-01.txt`, Internet Engineering Task Force, November 2001.

[23] D. Piscitello and J. Lawrence. Transmission of IP datagrams over the SMDS service. Request for Comments (Standard) RFC 1209, Internet Engineering Task Force, March 1991.

[24] D. Plummer. Ethernet address resolution protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware. Request for Comments (Standard) RFC 826, Internet Engineering Task Force, November 1982.

[25] D. Provan. Transmitting IP traffic over ARCNET networks. Request for Comments (Standard) RFC 1201, Internet Engineering Task Force, February 1991.

[26] J. Renwick. IP over HIPPI. Request for Comments (Draft Standard) RFC 2067, Internet Engineering Task Force, January 1997.

[27] J. Romkey. Nonstandard for transmission of IP datagrams over serial lines: SLIP. Request for Comments (Standard) RFC 1055, Internet Engineering Task Force, June 1988.

[28] rtnetlink, NETLINK_ROUTE - Linux IPv4 routing socket. Linux Man Page rtnetlink(7), April 1999.

[29] K. Sklower, B. Lloyd, G. McGregor, D. Carr, and T. Coradetti. The PPP multilink protocol (MP). Request for Comments (Draft Standard) RFC 1990, Internet Engineering Task Force, 1996.

[30] A. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *ACM Mobicom '99*, 2000.

[31] M. Stemm and R. H. Katz. Vertical handoffs in wireless overlay networks. *ACM Mobile Networking (MONET), Special Issue on Mobile Networking in the Internet*, 1998.

[32] J. Tourrilhes and C. Carter. P-Handoff: A protocol for fine grained peer-to-peer vertical handoff. In *Proceedings of the 13th IEEE International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC'02)*, 2002.

[33] J. Tourrilhes, L. Magalhães, and C. Carter. On-demand TCP: Transparent peer-to-peer TCP/IP over IrDA. In *Proceedings of the IEEE International Conference on Communications (ICC '02)*, 2002.

[34] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service location protocol. Request for Comments (Standards Track) RFC 2165, Internet Engineering Task Force, June 1997.

[35] X. Zhao, C. Castelluccia, and M. Baker. Flexible network support for mobility. In *Fourth ACM International Conference on Mobile Computing and Networking (MOBICOM'98)*, 1998.

Service-oriented Network Sockets

Umar Saif and Justin Mazzola Paluska
M.I.T. Laboratory for Computer Science
{umar, jmp}@mit.edu

Abstract

This paper presents the design and implementation of service-oriented network sockets (SoNS) for accessing services in a dynamically changing networked environment. A service-oriented network socket takes a high-level description of a service and opportunistically connects to the best provider of that service in the changing characteristics of a mobile system. An application states its high-level service requirements as a set of constraints on the properties required in a suitable resource and SoNS continuously monitors, evaluates and compares the available resources and (re-)connects to the resource that best satisfies the specified constraints.

Unlike content-based routing systems, SoNS is an end-host system, interposed at the session-binding layer, and offers connection-oriented semantics. SoNS' interface allows an application to tailor the planning policy used to establish and rebind a network session. SoNS is based on an extensible architecture to leverage the wide-range of emerging technologies for discovering and locating resources in a mobile system.

SoNS integrates a service-oriented abstraction with the traditional operating system interface for accessing network services, making it simpler to develop pervasive, mobile applications. We present an implementation for a mobile handheld device, analyze the performance of our system and describe an application to demonstrate the utility of our system.

1 Introduction

Advances in digital electronics over the last decade have made computers faster, cheaper and smaller. This coupled with the revolution in communication technology has led to the development and rapid market growth of embedded devices equipped with network interfaces. It has also promoted the development and widespread use of battery-operated portable computers, allowing users to carry their computation resources and tasks with them.

These advances have led to the recent activity in pervasive systems [1][2]. MIT's project Oxygen [22], and related pervasive computing projects elsewhere, aim to define computational environments that would allow users to carry their mobile handheld devices from one networked environment to another, possibly disconnected, environment while providing personalized ubiquitous access to services in the environment of the user.

Such a system must be able to continuously adapt to changes in user locations and needs, respond both to component failures and newly available resources, and maintain continuity of service as the set of available resources change. This requires more than service discovery [3] or simple content-based routing [4]; it necessitates a certain degree of planning involving continuous reevaluation of available alternatives, as well as heuristic compromises to best address the application's requirement using imperfect resources in the changing environment of the application [5].

Such opportunistic access to system resources is contrary to what is offered by traditional mobile systems [6] that aim to *preserve* access to a mobile host as the characteristics of the system change. Such systems do not cater to *context-aware* applications [5][1][2] that desire to access the best provider of a service (henceforth referred to as a *resource*) in their environment, rather than maintaining access to a particular host.

Traditionally, such a context-aware application must itself provide the planning involved in accessing the best available service-provider in its environment. Such applications typically contain a planning component that continuously reevaluates the available alternatives and provides access to the best available service-provider. These planning components often employ a resource discovery system to find the available alternatives and use the operating system socket interface to establish and rebind network connections as better alternatives become available. Most context-aware and adaptive applications layered on top of traditional operating systems and network routing architectures are examples of this model [7].

Where the above-mentioned model has the virtue that the application is free to use any arbitrarily complex planning policy befitting its requirements, allowing the underlying system to be policy-neutral, it requires every application to be capable of discovering, monitoring, evaluating and comparing the available alternatives in order to utilize the best available service-provider in its environment. In a pervasive computing environment, where such *opportunistic* access to service-providers is a *norm,* it is clearly desirable to separate this complexity in a *re-usable planning layer* that can be employed by different applications to opportunistically access resources in a dynamically changing networked environment.

Among the existing systems, the Intentional Naming System (INS) [4] comes closest to achieving this goal. The late binding architecture of INS allows an application to send *intentional datagrams* that carry a description of the properties of the required service, instead of the network address of a host, and an overlay of INS resolvers route these datagrams to the hosts that match the service description. Where this scheme of integrating service location and message routing alleviates an application from the task of continuously monitoring its environment and rebinding its network connections when a better alternative becomes available, INS provides limited planning for choosing the closest match to application requirements when more than one resource matches a service description. In this case INS simply relies on an application-level anycast to all the matching resources.

Even though it is conceivable that a more elaborate scheme could lead to more informed routing decisions, this approach of handling the dynamism of the system at the routing level inherently suffers from the following problems.

- The planning policy, used to select the best match to application requirements, is hidden from the application in the routing infrastructure and, worse, distributed in the network. Therefore, it cannot be tailored to suit the requirements of the various different applications found in a pervasive mobile system.
- Such content-based routing systems [4][8] only provide connection-less datagram semantics; every datagram carries the required service description which is resolved by, often an overlay of, network resolvers to deliver the message to an appropriate host. Therefore, such systems lack application-level session semantics, in that there is no concept of an application-level connection; two successive datagrams generated by an application can be routed to two different hosts, transparently to the application. This coupled with the characteristic fluctuations in the performance of wireless links and mobile hosts, means that an application has little control over which resource gets accessed, precluding applications with inherently connection-oriented semantics e.g. multi-media streaming applications. Such a system is also prone to *thrashing* between service-providers in the presence of frequent performance fluctuations and node failures.
- From a performance point of view, content-based routing, performed by resolving complex service descriptions at every hop in an overlay network, is considerably slower than traditional address-based network routing [4] since it introduces the cost of resolving a service description to a network address in the critical path of message delivery. Furthermore, including a full service description of the required service with every network message is wasteful of the scarce bandwidth available in a wireless network.
- Finally, content-based routing systems introduce a new API for network communication [4][21], which is often different from the traditional operating system interface, for accessing services in the system.

We propose Service-oriented Network Sockets (SoNS) to access services in a highly dynamic networked environment. A service-oriented network socket takes a high-level description of a service and opportunistically connects to the best provider of that service in the changing characteristics of a mobile system. An application states its requirements as a set of constraints on the properties required in a suitable resource and SoNS continuously monitors, evaluates and compares the available resources and (re-)connects to the resource that best satisfies the specified constraints.

Unlike content-based routing systems, SoNS is an end-host system, interposed at the session-binding layer, and offers connection-oriented semantics. Most importantly, SoNS allows an application to configure, and even replace, the planning policy used to evaluate and compare available alternatives and the semantics used for rebinding a network connection when a better alternative becomes available. SoNS integrates a service-oriented abstraction with the traditional operating system interface for accessing network services, making it simpler to develop pervasive mobile applications.

We favor this approach over a content-based routing scheme as it handles the dynamism of a mobile system at the stage of binding a network connection at an end-host, and hence 1) offers connection-oriented semantics 2) does not introduce the overhead of resolving a service description in the critical path of network communication, 3) does not require a service description to be carried with every network message, and 4) does not require any changes to the network routing architecture.

The rest of the paper is organized as follows. Section 2 identifies the design goals for SoNS and Section 3 describes the architecture of SoNS. Section 4 describes the operation of the SoNS constraint parser, section 5 describes the SoNS resource discovery framework, section 6 describes the architecture of the module used to evaluate resources and section 7 presents the support for network connection migration. In section 8 we describe the API exported by a service-oriented network socket and present a representative context-aware application built using SoNS. Section 9 describes the implementation of SoNS for a mobile handheld device, and section 10 presents performance analysis and evaluation. Section 11 describes related work and, finally, in section 12 we conclude the paper and outline future directions of our research.

2 Design Goals

In order to identify the goals for a system designed to provide opportunistic access to services in a dynamically changing system, we consider a simple example application of such a system.

In our example, a video-stream played by a user's handheld device is automatically redirected to the nearest display as she moves in an environment populated with displays, possibly from different vendors and conforming to different standards. In order to provide this *follow-me-video* functionality, the application requires opportunistic access to the nearest display of a decent size, located in the same subnet as the user. Furthermore, though the application requires access to a better display as soon as one becomes available, it would not like the video-stream to be switched between displays due to transient fluctuations in their access latency or when a display device is quickly carried past it by another user. Finally, the application must be notified before a session is migrated to a new resource so that, for instance, it can transfer some application-specific state to the new resource to resume access to the service or to even decline the rebinding suggestion all together.

In order to support such applications, our system must meet the following goals.

- **Resource Discovery and Selection**: The system must be able to discover resources based on a high-level service specification. Additionally, the system must define a planning framework capable of evaluating and comparing the properties of available alternatives in order to find the closest match to application requirements.
- **Expressiveness:** An application must be able to state its requirements such that they can be used for both discovering and, subsequently, comparing the suitability of available alternatives. An application must be able to state the attributes required in a suitable resource, the range of acceptable values for each attribute, the preferred values for an attribute and the relative importance of each attribute to the application.
- **Extensibility:** In order to support a diverse set of applications in a variety of network characteristics and standards, the system must not enforce any fixed policies that could limit the use or efficacy of the system. Instead, the system must define an architecture that may be extended to handle different application requirements, network characteristics and standards.
- **Connection Rebinding Semantics:** It must be possible for an application to configure the semantics of rebinding a network session when a better alternative becomes available. Based on our target applications, we identify the following parameters to provide an application with the flexibility to configure the semantics of session rebinding.
  - **Context** It must be possible for an application to configure the context within which it wants to find the best resource for its requirements e.g. current subnet, current room.
  - **Agility**: It must be possible for an application to configure the agility with which it wants the system to react to *valid* changes in its context.
  - **Hysteresis**: It must be possible for an application to configure the hysteresis of the system, indicating how long the system should wait before reacting to a change, in order to avoid reacting to transient fluctuations that are not of interest to an application, and to protect against *thrashing*.
  - **Application-notification:** It must be possible for an application to register a call-back method, which is invoked by the system to notify the application about the availability of a better alternative. This notification can be used by the application to prepare for the rebinding of the network session. It must also be possible for the application to decline the suggestion of rebinding the session to the new resource.
- **Performance:** Where the system must include a planning function capable of evaluating and comparing a set of resources competing against application requirements, this planning task must be fast enough to quickly respond to changes in the system. Furthermore, as our system is interposed at the operating system socket level, it must be comparable in performance with the traditional socket-based communication. Finally, it must not introduce an overhead for applications that do not require service-oriented communication.

2.1 Service-oriented Network Sockets

Our service-oriented network session layer includes an attribute-based discovery framework for discovering resources in the system, as well as an *evaluator* module for computing the suitability of available alternatives against application requirements.

Since a network socket provides a portal between an application and the network communication support of an operating system, it presents a natural interface for incorporating application-level policies for establishing a service-oriented network connection by discovering and evaluating the available alternatives.

Service-oriented Network Sockets offer an additional socket domain that takes a high-level service specification as the destination name, instead of a network address, and defines additional socket options to configure the rebinding semantics for the service-oriented session. Using this interface, applications configure a network socket with an appropriate context, agility and hysteresis, and *connect* the socket by

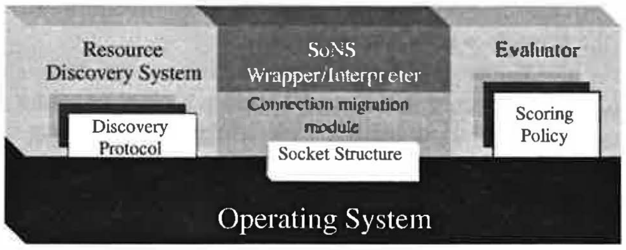


Figure 1: SoNS System Architecture

providing a service description to open a service-oriented network session. Using these application-level semantics, SoNS locates the most appropriate resource in the given context and establishes a network connection. If any subsequent changes in the system render another resource more suitable for application requirements, in accordance with the agility and hysteresis semantics of the application, SoNS notifies the application and migrates the session to the better alternative.

A service description is expressed as a set of constraints on the properties of an acceptable resource. As opposed to the resource discovery systems that find a resource by performing an exact pattern-match on its attribute-value pairs [3][4], the use of a constraint language in SoNS, for stating an evaluation criteria, offers the flexibility to evaluate and compare the alternatives available in a given context in order to find the closest match to the requirements of an application.

The design of SoNS handles the heterogeneity of discovery standards and application requirements by using a modular and extensible architecture for resource discovery and evaluation. Protocols for discovering resources and the policy for evaluating available choices can be tailored according to the application requirements and discovery standards used by different resources.

By handling the dynamism of the system at an end-node, SoNS does not require any changes to the network routing infrastructure. Therefore, as opposed to systems that employ application-level content-based routing [4] to address the dynamism of the system, SoNS architecture does not introduce extra routing complexity in the participating nodes, achieves better performance, and leverages the underlying network support for quality-of-service.

3 System Architecture

Figure 1 shows the architecture of a Service-oriented Network Sockets system. In order to facilitate application-specific extensibility, portability, accounting and fault-isolation, Service-oriented Network Sockets are implemented as a user-space wrapper around a traditional socket interface, instead of as a kernel module.

The SoNS architecture has four components: a resource discovery module, an evaluator module, a connection migration module, and a socket-wrapper module. Below we describe these modules in detail.

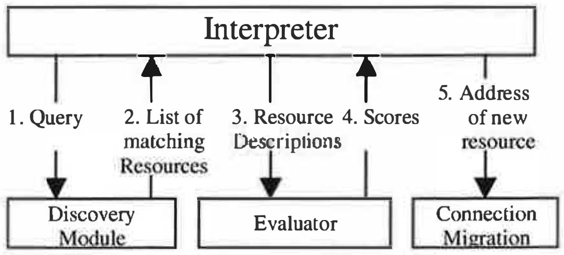


Figure 2: The SoNS Interpreter drives the different components in the system

3.1 SoNS Interpreter

The SoNS Interpreter, shown in figure 2, lies at the heart of the system and drives the different modules of the SoNS architecture; it parses the constraints specified by an application, discovers matching resources by invoking the resource discovery module, invokes the evaluator module to evaluate the suitability of any matching resources, and finally, in the case when a new resource becomes a better choice for the application, notifies the application and requests the connection migration module to migrate the connection to the new resource.

In order to allow this processing to be accounted on a per-connection basis, SoNS system forks a new Interpreter for every service-oriented network socket created by an application.

3.2 SoNS Interface

SoNS is designed as an extension of the operating system socket interface; it implements all the methods and options of a traditional AF_INET Unix socket, with additional options for establishing service-oriented network connections.

A service-oriented network socket extends a traditional network socket in the following ways:

1) The call to create an operating system socket accepts an additional domain, *AF_SONS*, for creating a service-oriented network socket. AF_SONS extends an AF_INET socket and allows an application to choose between (sock_stream) and UDP (sock_datagram) as the transport protocol for a service-oriented session, including support for the various options associated with these transport protocols e.g. TCP_NO_DELAY for TCP.
2) The *connect* method of a service-oriented network socket takes a high-level *service description*, instead of a network address, to establish a service-oriented network session. The service description is expressed in a simple constraint language, described in detail later in the section.
3) A service-oriented network socket can be configured with four additional options (as arguments to *setsockopt), context, agility, hysteresis and*

```
ConstSpec = Nested | Cmplx | Smpl
Nested = Cmplx (Cmplx+)
Cmplx = (Logical (Smpl  Smpl+))
Smpl =  (Relation Attribute)  |
    (Relation Attribute Range | Value) |
    (Relation Attribute Range | Value) Weight
Logical = AND | OR
Relation = < | > | =
Attribute = String
Weight = Integer
Range = Numeric Numeric
Value = String | Numeric
String = [a-z]+[a-z1-9]*
Numeric = Integer | Float
Integer = [1-9]+
Float = [1-9]+.[1-9]*
```

Figure 3: Constraints Language for SoNS

application-callback, to tailor the session rebinding semantics according to application requirements.

4) Finally, when configured with the optional application-callback, a service-oriented network socket invokes a callback method to notify (and seek permission of) the application before rebinding a network connection to a better alternative.

3.3 SoNS Constraint Language.

Though previous resource discovery systems offer varying degrees of sophistication for looking-up resources based on their attributes [9][4], these systems do not offer support for evaluating and comparing the suitability of matching resources against application requirements. SoNS, on the other hand, allows applications to specify the criteria for discovering, evaluating and comparing the available alternatives as a set of constraints expressed in a simple constraint language.

Though several sophisticated constraint languages have been proposed in other problem domains [10], the constraint language used to express a service-requirement in the SoNS system achieves a delicate balance between the expressiveness required for evaluating the suitability of available service-providers and the simplicity of design necessitated by the paucity of resources available in a mobile device.

The grammar for the SoNS constraint language is shown in figure 3. An expression in the SoNS constraint language lists the attributes that must be present in the selected resource, along with a range of acceptable values for each attribute. In order to define an evaluation and comparison criterion, a constraint also includes an operator, (less-than "<", or greater-than ">"), to indicate the preferred extreme in the range of acceptable values; resources with attribute values closer to the preferred extreme are favored over the resources with values further away towards the other extreme. This approach of allowing an application to express its requirement as a range of acceptable values, instead of a single scalar value, has the following merits. 1) It provides the flexibility to satisfy the requirements of an application with imperfect resources in its environment 2) It provides the system with a yardstick to compare and evaluate the matching resources against application requirements. 3) It encourages an application to explicitly declare its *scale of tolerance* for an attribute value; a change k in a range L $\leftrightarrow$ K is more significant than the same amount of change k in a larger range, L $\leftrightarrow$ (K +P).

```
(and (= device display)
     (> (size 15 30)
     (= color yes)
     (or (> video-streams 1)
          (= load 0)))
```

Figure 4: An example constraint specification expressed in the SoNS constraint language

In the case where an application is interested in the *least* or the *greatest* value for an attribute, irrespective of the specific value of the attribute, the application can leave the range unspecified. This could be used by an application to, for example, connect to the least loaded server in its environment, expressed as "< load".

SoNS also allows open ended ranges in the case where the application is interested in having an attribute value to be greater than (or lower than) a certain threshold, but perceives no marginal gain as the value of the attribute moves further away from the specified threshold. SoNS handles this case by treating the unbounded end of a range as 0 or a large positive integer, depending on which side of the range is unspecified.

Not all attributes of a resource required by an application are of the same importance to the application. SoNS handles this requirement by allowing an application to specify the relative importance of the listed attributes by attaching a (integer) weight with every attribute; an attribute with a weight of 4 is twice as important to an application as an attribute with a weight of 2.

Attributes that are allowed to have only a single value, including the attributes with textual values, use an equality ("=") operator and do not specify a range or attach a weight to the attribute; a resource description that does not match an equality constraint is simply rejected. Attributes that must be present in a matching resource, but whose value is not of interest to the application, are specified as a don't care value, stated as ANY.

Finally, the constraint language includes two logical operators, *conjunction* and *disjunction*, to allow individual constraint-expressions to be combined into a *composite constraint specification.* A composite constraint specification can have a hierarchical structure; constraints can be grouped (associated) and nested using braces, and the logical operators are distributed over nested constraints when evaluating a constraint.

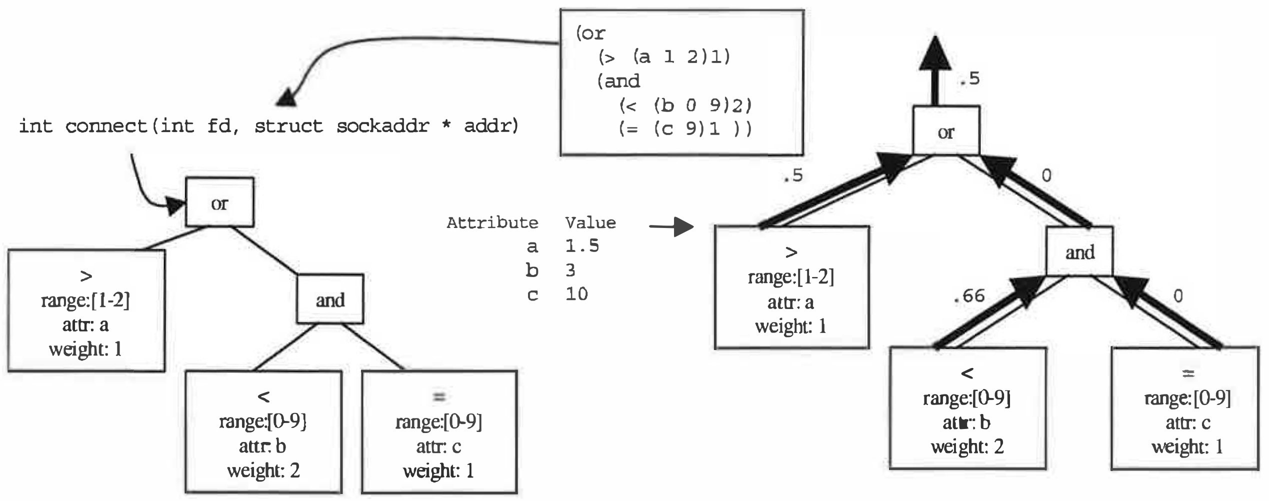


Figure 5: An illustration of constraint parsing and evaluation by the SoNS Interpreter

To illustrate the expressiveness of the SoNS constraint language, we show how the requirements of a follow-me-video application, presented in section 2, will be expressed in our language. Such an application can impose the following constraints on the display used by it. 1) The display must be more than 15 inches in size, for clear viewing, but less than 30 inches, due to the resolution limitations of the video-encoding scheme, 2) it must be capable of rendering colors, 3) and should be either capable of displaying more than one video-stream simultaneously or must not be in use. These requirements would be expressed in the SoNS constraint language as shown in figure 4. It is worth noting that the use of an open-ended range for the number of video-streams supported by the display device implies that the application is indifferent to the number of streams being displayed on the screen. If the application prefers to use a less cluttered screen, it will provide a closed range, and will use the "<"operator to indicate that a display capable of showing fewer streams is preferable. Therefore, the use of a range to express a constraint, in fact, encourages an application to be more precise in defining the, often assumed, precincts of context-awareness.

3.4 Semantics of Session Rebinding

Besides the constraints specified by an application to define the criteria for comparing available resources against application requirements, SoNS also allows an application to tailor the semantics of rebinding the network session by controlling the parameters for detecting and reacting to changes in the system. A service-oriented network socket takes four additional options as arguments to the *setsockopt* library call.

Context: An application can specify its *context* as a sub-net address, location of the looked-up resources, number of network hops traversed by a discovery message or any other metric meaningful for the discovery protocols part of the SoNS architecture. For example, the current implementation adjusts the *SCOPE* of an SLP [9] network query to limit the context of the discovery.

Agility: An application can specify the *agility* with which it reacts to changes in the system by adjusting the frequency to probe the system for changes. The agility is specified as the interval between successive probes, stated in seconds.

Hysteresis: An application can keep the system from reacting to transient changes, not of interest to the application, by specifying a value for *hysteresis.* The hysteresis is stated in terms of the number of probes for which an application requires the properties of the resources in its context to be consistent before SoNS (notifies an application and) switches the connection to a better alternative.

Application-Callback: Finally, an application can register a callback with the socket, which, if registered, is used to notify the application when a better alternative becomes available. This notification, parameterized with the description (including the network address) of the new resource, can be used by an application to prepare itself to switchover to the new resource or to reject the change by returning a false value from the callback. It is worth noting that since a connection migration can only happen when the application returns control from the call-back, the application can use the call-back to delay the migration to a "migration-safe" point in its control flow.

4 Constraint Parsing

The constraints specified by an application are used both for *discovering* and *evaluating* resources in the context of an application. To accomplish this, the constraints are parsed into a tree data-structure, which serves as an in-core representation of the application requirements for discovering and evaluating resource descriptions.

Constraints are read as a plain-text string from the sockaddr_sons structure passed by the application in a connect()socket call (refer to figure 5). The string is then parsed using a standard GNU Flex/Bison lexer/parser into a constraint tree. The parser makes a distinction between composite constraints and simple constraints. Simple constraint, specifying a range over a single attribute, are placed at the leaves of tree, while composite constraints, containing nested constraints composed by taking disjunctions (OR) and conjunctions (AND) of simpler constraints, are represented at the intermediate nodes of the tree (refer to figure 5).

The parser also fills-in any missing bounds, 0 for less than constraints and a large integer for greater than constraints, as well as missing weights with a default of 1.

5 Resource Discovery

After constructing a constraint tree, the SoNS interpreter invokes the discovery module with the list of attributes at the leaves of the constraint tree. The discovery module invokes the discovery protocols registered with it and returns the matching resource descriptions to the interpreter.

The interpreter then passes this list to the evaluator module, which assigns each resource a score by comparing the values of its attributes against the constraints stored in the constraint tree. The evaluator invalidates the resource descriptions with attribute values outside the range specified by the application, as well as the resources that fail to meet an equality constraint.

After the initial setup, this procedure is repeated every time the probe period specified by the application expires. An application can also force a probe/evaluate cycle, for instance on the command of a user. After receiving the score for each resource, the interpreter removes all the resource descriptions that were rejected and forms the “n-best-list” for the probe. If the application forced the probe (by invoking *connect* on an already connected socket), then the resource with the highest score is chosen from the n-best-list and the socket is migrated to its network address (just like the initial setup). However, if the probe was a normal periodic probe, the system enters the *hysteresis* phase. In the hysteresis phase the n-best-list from one probe/evaluate cycle is compared to the n-best-list stored from the previous cycle and the resources present in both new and old probes have their hysteresis value increased by one. Resource(s) with a hysteresis value greater than the hysteresis value specified by the application are separated and the connection is migrated to the network address of the resource with the highest score. In the case where an application has registered a call-back, SoNS invokes the callback method, with the description of the chosen resource, before performing the migration, and migrates only if the application-callback returns a true value (indicating application’s approval of the connection migration). Upon migration of the network connection, the n-best-list is reset and the process is started anew.

5.1 SoNS Resource Discovery Framework

Our target network environment often comprises of resources conforming to different resource discovery protocols, e.g. IETF SLP [9], INS [4] and SSDP [11], due to both commercial and technical reasons. Therefore, a service discovery framework based on just a single discovery protocol is not sufficient to discover the various resources found in a pervasive mobile system.

SoNS handles this heterogeneity by defining an extensible resource discovery framework, capable of employing different discovery protocols to discover resources in the system. A discovery protocol is added to SoNS by registering a pointer to its *look-up* method, while SoNS performs resource discovery by invoking the look-up methods of all the discovery protocols registered with it.

However, various discovery protocols found in our target environment offer different degrees of expressiveness for looking-up resources in the system. Protocols like INS [4] and SSDP [11] simply take a list of attributes and match them with the attributes of the resources being looked-up, whereas more sophisticated protocols like SLP [9] and SSDS [3] can perform complex queries containing conjunctions and disjunctions on nested lists of attributes, as well as range comparisons for attributes with numerical values. In order to interoperate with such diverse protocols, SoNS translates a service specification to a very basic query format common to all discovery protocols.

SoNS resource discovery framework invokes a constituent discovery protocol with a simple list of ASCII-encoded attribute names, constructed by taking the attribute names from the leaves of the constraint tree created by the SoNS parser. Upon invocation, a discovery protocol finds the resources containing the specified attributes, and returns their descriptions in a list of *feature-sets*: sets of attribute-value pairs. The matching resource descriptions, encoded as feature-sets, are passed on to the evaluator module to evaluate their suitability against the constraints specified by an application.

It is worth noting that, in order to achieve compatibility with simpler protocols, this scheme does not require any filtering involving value comparisons to be performed by a discovery protocol. Rather, discovery protocols look-up resources by simply performing a pattern match on the specified attributes, and the suitability of a resource, based on the values of the looked-up attributes, is computed in the SoNS evaluator module.

Passing a query as a simple ASCII-encoded list of required attributes also has the virtue that it can be easily converted to a more ornate format, by a simple wrapper around the lookup interface, if required by a more sophisticated discovery protocol.

5.2 Context of Discovery

Along with a pointer to a look-up method, a discovery protocol can also register a pointer to a method for setting the scope of the network queries generated by the discovery protocol. This method is invoked by SoNS when an application specifies a context of interest as an option to a service-oriented network socket. For example, SLP and SSDS register a pointer to a method that sets the value of SCOPE of the discovery agent to configure the context of the network queries. Though some simpler protocols, e.g. SSDP, lack support for scoped queries, and hence, do not register this method, we believe that such support is the key to the scalability of a pervasive discovery protocol and will soon find its way in mainstream discovery protocols.

5.3 Probing vs. Advertising

A mobile device wishing to discover resources in its environment can either *passively* listen to advertisements by other resources in the system or can *actively* probe the network with periodic discovery messages.

SoNS uses active probing as it makes it simpler to support application-level semantics for session-rebinding. Applications configure the session rebinding semantics by setting 1) the frequency of probing, to adjust the agility with which resources are discovered, 2) the scope of a probe message, to adjust the discovery context and 3) the number of probes for which the properties of a resource must be consistent, to set the hysteresis of the system.

We favor probing over advertisements because in an advertisement based system the scope and frequency of the messages generated by a resource to advertise itself to the system cannot be adjusted to suit the requirements of any single application. Furthermore, with resource advertisements arriving asynchronously at different frequencies from various resources, there is no clean way to specify the hysteresis of the system.

From a design point of view, in an advertisement-based system, where resources are required to continuously advertise themselves to the system in the hope that some application might be interested, introduces a continuous overhead of network messages and processing of advertisement messages even when there is no application listening to the advertisements.

Finally, probing is supported by all the resource discovery protocols found in our target environment (though some protocols can also be configured to operate in an advertisement-based mode).

5.4 Directory-based versus Peer-to-peer Discovery

Resources can either respond to queries directly, in a peer-to-peer setup, or could register their descriptions with a directory service which could be searched to locate resources.

SoNS' extensible design does not impose a restriction on which of the two methods is employed by a constituent discovery protocol to discover resources in the system. However, we believe that a peer-to-peer model is more suitable for supporting application-specific session rebinding.

Though a directory-based setup avoids query broadcasts, and, hence, presents a more scalable design, it suffers from the following limitations in a dynamically changing system. 1) A directory-based architecture depends on the availability of host(s) in the system that are capable and willing to answer queries on behalf of other resources. 2) A directory-based scheme introduces the overhead of keeping the directory state consistent with the (oft-changing) properties of resources in the system. 3) The directory service can itself cause a bottleneck in the system. Since in a peer-to-peer setup resources themselves report their, up-to-date, properties, the rate of probing provides an accurate mapping for the rate of adaptation expected by the application; this can only be guaranteed in a directory based system when the directory service is always consistent with the changes in resource properties.

6 SoNS Evaluator Module

The discovery framework returns all the matching resource descriptions returned by the various discovery protocols to the interpeter, which passes these resource descriptions to the evaluator module. The SoNS evaluator module performs the planning required to select the resource, among the available alternatives, that comes closest to satisfying the service requirements of an application.

To motivate the evaluation strategy used by the SoNS evaluator module, consider a situation where the follow-me-video application mentioned above moves into an environment with two displays: one closer to the handheld device but the other larger in size and with better resolution. In this situation, there is no clear winner (that is better than all the other available alternatives in every aspect). A naïve solution could be to count the number of attributes for which a resource "beats" other alternatives and pick the resource with the maximum number of "wins". However, such a solution not only leads to a combinatorial explosion but also requires every sample of attribute values to be kept for later comparisons according to application's hysteresis requirements.

SoNS evaluator is designed to be simple and responsive to changes and does not require the attribute values of every resource to be preserved across multiple probes. SoNS achieves this by using a simple scoring scheme which sums-up the suitability of a resource in a single scalar value for efficient comparisons.

SoNS evaluator takes a list of feature-sets, along with the application's constraint tree, and returns a corresponding list of positive integer scores reflecting the suitability of each resource. A resource with an attribute that fails to meet an equality constraint or has a numerical value outside the range specified by an application is assigned a score of zero.

Figure 5 shows the operation of the default SoNS evaluator. SoNS' default evaluator performs a depth-first search of the constraint tree. On reaching a simple constraint at a leaf node, it extracts the value of the corresponding attribute from the resource's feature-set and compares the value with the range specified in the constraint. If the value satisfies the constraint, then a score between 0 and 1 is calculated based on where the value falls in the valid range. If the constraint specifies that smaller is better, then a value equal to the lower bound is assigned score 1 and a value equal to the upper bound is assigned score 0, with all other values being assigned linearly within that range. The reverse occurs for constraints indicating that larger values are better. If the constraint specifies only equality or the ANY keyword, then any value fitting the constraint is given a score of 1. Finally, the score is multiplied by the constraint's weight and returned as the value of that leaf node.

After assigning scores to the leaf nodes, scores for the intermediate nodes, containing conjunctions and disjunctions, are calculated using the following algorithm. An OR node acquires the score of a child node with the highest score in its sub-tree, while a score of zero is assigned if all of its children nodes have a score of zero. An AND node is evaluated in a complimentary way: the score of an AND node is calculated by adding the scores assigned to its child nodes, while any child node with a score of zero causes the AND node to be assigned a score of zero. The overall score of a resource is the score calculated for the root of the constraint tree using the attribute values in the resource's feature-set.

We have found this simple evaluation strategy to be sufficient for our purposes for the following two reasons. 1) It keeps the design of SoNS simple enough to be hosted in resource constrained mobile devices and 2) the simplicity of the algorithm used for evaluating and comparing the available alternatives incurs minimal penalty in terms of the responsiveness of the system; where a more elaborate scheme could be used for comparing the suitability of available alternatives, it would increase the time spent in evaluating a resource, resulting in an increased latency between the time a viable resource become available and when the system recognizes its superiority.

As described earlier, the scores returned at each probe are compared by the Interpreter according to the hysteresis semantics of the application and a winner is chosen if a resource consistently scores better than other resources.

The extensible design of SoNS also allows the default evaluation policy to be replaced by more efficient or specialized algorithms better suited to individual application requirements. An application can replace the default evaluation policy by registering a pointer to an application-specific evaluator with the SoNS evaluation module. This allows more involved constraint satisfaction engines, for example as proposed in [12], to be employed for calculating the relative utility of available resources. Such planning and constraint satisfaction systems are a topic of our current research.

7 Connection Migration Module

Once a better resource has been selected, the SoNS Interpreter requests the connection migration module to migrate the network connection to the new resource.

The semantics of migrating the network connection from one resource to another depend on both the stateful-ness of the service being accessed and the reliability guarantees offered by the underlying message transport protocol [13]. Migration of an unreliable network connection to a stateless service is accomplished by simply closing the old network connection and opening a fresh connection to the new resource. However, additional support is required for migrating reliable connections and for managing stateful services [13]. Migration of a reliable connection requires support for preserving the sequence of messages across migration, while a connection to a stateful service can be migrated transparently across resources only when the state accessed at the old resource is also available at the new resource in the form that the access to the service can be resumed at the new host from where it left-off at the old resource.

The former requires a reliable transport protocol with support for migrating an active connection, while the later also requires a system for distributing and maintaining consistent state across replicated instances of a stateful service.

This paper focuses on enabling a client to utilize the best provider of a service in its changing context; the subject of replicating and synchronizing stateful services has been extensively researched by others [14] and is not covered in this paper.

SoNS uses the Migrate system [15] for migrating network connections between resources. We chose the Migrate system as it provides support for securely migrating both reliable and unreliable network connections, as well as a lightweight, soft-state based consistency management system to support connection migration across stateful servers. Unlike other connection migration systems, like SCTP[16], that require the network addresses of all the potential servers to be known at connection setup time, Migrate allows a connection to be migrated to a newly available server using the TCP migrate options. Having said this, the modular design of SoNS allows other connection-migration systems to be used as well, though we have not integrated other such systems with SoNS as yet.

8 Applications

This section describes the API of SoNS and a simple, yet representative, application we have developed to demonstrate the utility of service-oriented network connections offered by the SoNS architecture.

Our test applications were developed for a Compaq iPAQ, fitted with a backPaQ and running familiar Linux. Our backPAQ is fitted with an 802.11b wireless card, video-camera, accelerometer and the Cricket Location detection system [17].

8.1 Follow-me-video

We have used SoNS to develop a *follow-me-video* application. A follow-me-video application running in a handheld device carried by a user re-directs the video stream to the display closest to the user as she moves in the system. In our test environment, all resources (server devices) are also fixed with Cricket Beacons to measure their distance relative to other Cricket-enabled devices (including our handheld device).

The relevant code snippet from our example application, mentioned in section 3.3, is shown in figure 6. Our example application generates an MPEG 1 encoded stream and is interested in the nearest display with 1) Resolution: 640x800 – 1280x1600 (with preference for displays with higher resolution), 2) Size: larger than 15 inches to allow viewing from a distance, but less than 30 inches due to the limitation of the encoding resolution (with preference for a larger display)

The application creates an AF_SONS domain socket, and specifies the following options: Context: 6th-Floor, Agility: 5 (seconds between probes), Hysteresis: 3 (number of probes), Call-back: pointer to method that forced a base frame to be transmitted. The application then connects the socket by giving it a composite constraint specification for the properties of the display device.

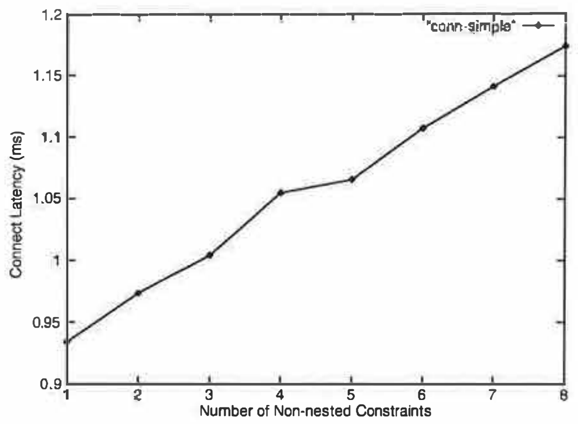
SoNS sets the SCOPE of the SLP user agent to 6th-Floor, probes the network for display devices, and connects to a display that scores the highest points among those present on the 6th floor. The distance to a display device is measured by invoking the Cricket location system (added to the system just like another discovery protocol). After making the initial connection, the network is probed, using SLP and

```
int main(int argc, char ** argv) {
  int sockfd;
  int intopt;
  char * charopt;
  size_t opt_sz;
  sons_callback_t cbsons;
  struct sockaddr_sons sasons;
  sockfd = socket(AF_SONS, SOCK_STREAM, 0);
  intopt = 5;
  opt_sz = sizeof(int);
  setsockopt(sockfd, SOL_SOCKET,
          SO_PROBE_PERIOD, &intopt, opt_sz);
  intopt = 3;
  opt_sz = sizeof(int);
  setsockopt(sockfd, SOL_SOCKET,
          SO_HYSTERESIS, &intopt, opt_sz);
  charopt = "floor-6";
  opt_sz = strlen(charopt) + 1;
  setsockopt(sockfd, SOL_SOCKET,
          SO_SERVICE_SCOPE, charopt, opt_sz);
  cbsons = force_b_frame();
  opt_sz = sizeof(sons_callback_t);
  setsockopt(sockfd, SOL_SOCKET,
          SO_CALLBACK, &cbsons, opt_sz);
  charopt =
          "(and (= service display)\n"
              "(= media mpeg1) \n"
              "(> xresolution 800 1600) \n"
              "(> yresolution 640 1280)) \n"
              "(> displaysize 15 30) \n"
              "(< distance))";
  sasons.sin_family = AF_SONS;
  memcpy(sasons.query, charopt,
       strlen(charopt));
  connect(sockfd, (struct sockaddr *) &sasons,
        sizeof(struct sockaddr_sons));
 //...read and write using standard socket
 // calls...
  close(sockfd);
  return 0;
}
```

Figure 6: Code snippet from the follow-me-video application developed using SoNS.

Cricket, every 5 sec and available resources are compared according to the hysteresis. If a better display becomes available, SoNS invokes the application callback, which forces a base MPEG frame to be transmitted upon migration, so that the playing of video at the new display can be resumed without jitter.

Our example application also monitors the accelerometer embedded in the backPAQ to find out if the device is moving and with what speed. If the application discovers that the handheld device is mobile, the application can increase the rate of probing and reduce the hystersis value, according to the degree of movement reported by the accelerometer, in order to take advantage of the displays that become available for a short time when, for example, a user walks down a hallway.



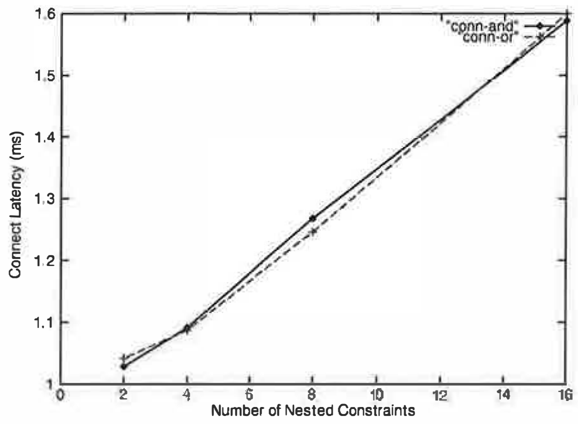


Figure 7: The cost of a connect() call (averaged over 100 tests) increases only linearly with the number of constraints.

9 Implementation

SoNS was implemented in GNU C/C++ on GNU/Linux. The code is divided into four modules: the socket interface library, SoNS interpreter, wrappers for the resource discovery protocols, and the evaluator module.

9.1 Socket Library

The socket library code overrides the socket interface to offer the extended SoNS interface, and spawns an interpreter daemon to periodically discover and evaluate available resources.

In order to provide an extended interface, we either needed to modify the underlying Linux libc or use a package that captures system calls and redirects them through other functions. We chose the latter route and specifically chose to use TESLA [18]. TESLA allows arbitrary handlers to be inserted between an application's socket call and the underlying socket kernel calls, precisely matching our needs. Furthermore, the Migrate architecture, which we use for connection migration, also uses TESLA, so the overhead of using TESLA would be present in our system anyway.

We wrote TESLA handlers to override the calls to socket(), connect(), getsockopt(), setsockopt(), and close() functions. The most interesting overridden call is connect(). The connect call exercises the entire system since it sends a message to the interpreter daemon telling it to force a network probe, then picks the best service, and finally calls connect on the underlying socket structure. getsockopt() and setsockopt() simply update socket-related data in the daemon. The interpreter daemon itself merely sits in a loop waiting either for an event from the TESLA handler or an alarm signal indicating that it should perform a periodic poll/evaluate cycle.

9.2 Resource Discovery Protocols

The interpreter invokes the *run_query* method of all the discovery protocols registered with it. The run query method takes an array of required attribute names and returns a list of attribute-value bindings. This method is the sole interface between the interpreter and the discovery protocols so that discovery protocols can be easily added/replaced.

The current implementation employs two discovery protocols to find resources in the system: IETF Service Location Protocol [9] and the Cricket Location System [17] (for estimating distance to available resources). We use the OpenSLP implementation of SLP, with the User Agent configured to perform discovery in a peer-to-peer fashion, by multicasting the query on the SLP multicast channel.

10 Performance Analysis and Evaluation

Unlike content-based routing systems, the session-oriented approach of SoNS moves the cost of resolving service descriptions from the critical path of a network message delivery to the stage of establishing and, subsequently, rebinding a network session. Therefore, we evaluate the performance of SoNS by measuring 1) how quickly it can setup a service-oriented network session and 2) how quickly it can rebind the network session when a better alternative becomes available.

All tests were performed on a Pentium III with 256MB of RAM running Linux 2.4. Since we wanted to isolate our system from network latency, we used an in-memory stub SLP rather than the OpenSLP SLP. We expect most constraint specifications to have between 1 and 15 elements and be a combination of both simple constraint specifications and composite constraint specifications. Our tests span this space: we vary the number of attributes in a straight-line set of simple constraint specifications and also vary the height of the tree of composite constraint specifications.

10.1 Session-setup Latency

The SoNS system adds latency in two places, at a socket() call where we fork a daemon and initialize all the discovery protocols installed in the system, and on a connect() call where we must decide which device is the best device available.

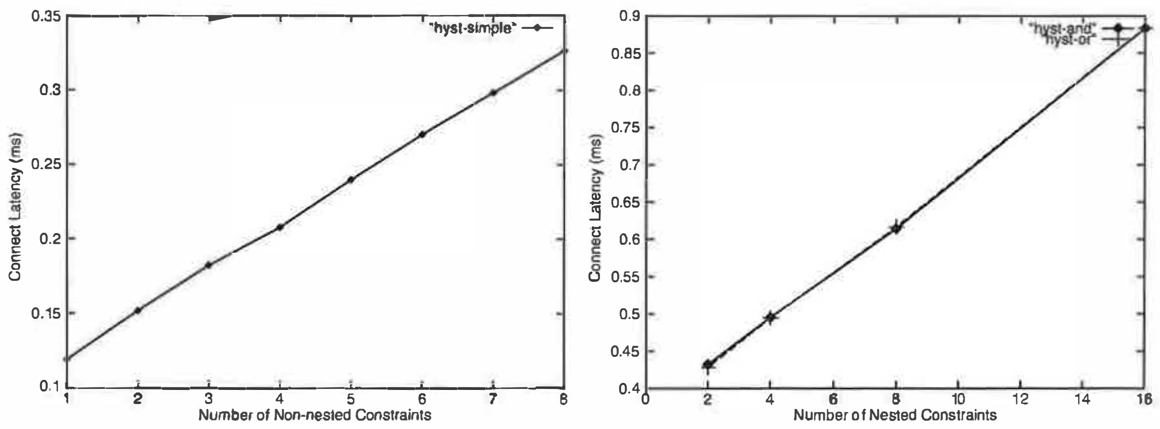


Figure 8: Latency (averaged over 100 tests) between the time a better alternative becomes available and the time SoNS realizes its presence (and signals the application about its presence).

In all of our tests, the *socket* call took between 2ms and 5ms. This is fairly high, but represents the cost of forking a process and all inter-process communication between the daemon and the SoNS TESLA handler.

Figures 7 summarizes the cost of a connect call as we varied the number of constraints. The cost of the connect call increases only linearly as the number of constraints, both nested and non-nested, are increased. The connect latencies in our system hover around 1ms, which, though higher than expected, is acceptable when amortized over the life of the connection.

10.2 Session-rebinding Latency

Another form of latency shows itself as the time between the discovery of a resource and signaling the application that the SoNS system has found a new best resource. Ideally, this should be zero, but we must evaluate the services returned, which has a non-zero cost. Figure 8 show the time elapsed from when a call to run_query() method returns — with matching resources on the network— and when SoNS invokes the application-callback to notify the presence of a better alternative (given the hysteresis semantics). This latency again increases only linearly with the number of constraints (both for nested and non-nested constraints), and more importantly, hovers only around 200-500μs of latency.

In order to achieve compatibility with simpler protocols, SoNS does not require any filtering involving value comparisons to be performed by a discovery protocol. Instead, discovery protocols return all those resources that contain the required attributes and the evaluator module performs the value comparisons to compute the suitability of matching resources. However, since all the value comparisons in this scheme are performed by the evaluator module, the evaluator must be able to efficiently compare a moderately large number of matching resource descriptions. Figure 9 shows the time spent in the evaluator module as we increased the number of resource descriptions processed by the evaluator. This cost increases only linearly and hovered only between 0.8 – 1.0 msec in our tests.

The simplicity of our system makes it suitable for mobile handheld device. The memory footprint of our system varied between 0.8 MB to 1 MB during our experiments.

10.3 Evaluation

Where our system achieves acceptable performance, we found that the following implementation choices incurred unwanted overhead:

- Per-socket interpreter daemon *processes,*
- Use of standard IPC between the socket wrapper and the per process interpreter deamon, and
- fixed-point arithmetic.

Our implementation would be faster if we were able to communicate with the daemon without copying through interprocess communication channels. This could be accomplished by using a thread library at the cost of making our code less portable. Secondly, we maintain a separate daemon process for each socket to allow for fine-grained accounting. However, this is inefficient compared to an implementation in which a single deamon process handles the discover/evaluate cycles for all the sockets, since such an implementation would save the cost of spawning a daemon every time a socket is created, and might allow for optimizations by batching queries by different applications.

Finally, our system is slowed down by the use of the fixed-point math system we wrote for computing resource scores. This is because the iPAQs we include in our target platforms do not have floating point units and incur an order of magnitude performance hit on floating point performance. Since scoring and weighting is an inherently floating-point process, we were forced to write our own, non-optimized, implementation of fixed-point arithmetic. We are currently working on a more efficient implementation in the light of these observations.

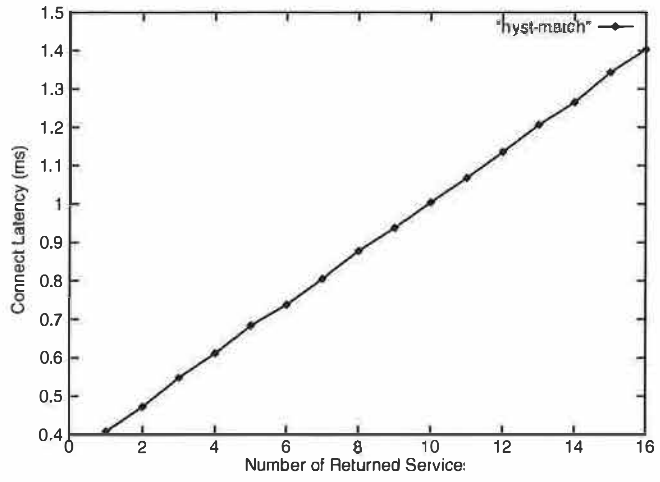


Figure 9: The hysteresis latency increases linearly with the number of services returned by the SoNS discovery framework.

11 Related Work

SoNS integrates a service-oriented abstraction with a traditional operating system communication interface. Using SoNS, applications open a network connection with an abstract service specification, instead of a network address, and the system automatically connects the application to the most suitable server in its changing environment. SoNS combines resource discovery and evaluation with a connection migration system to provide application-specific opportunistic access to service providers. SoNS' modular architecture can be extended with various resource discovery, location-detection, and connection migration protocols, and its evaluation policy can be customized to individual application preferences.

Therefore, SoNS is designed to complement and leverage recent research in resource discovery, location-detection and connection migration protocols, not to replace such systems. In fact, SoNS was motivated by the desire to combine pervasive mobile computing technologies developed for MIT's Project Oxygen, e.g. INS, Migrate and Cricket, to leverage context-aware applications in a mobile handheld device.

The recent interest in pervasive computing environments has given rise to a proliferation of systems that allow resources to be dynamically discovered based on their attributes. As opposed to the white-pages style lookup offered by systems like DNS that simply resolve a resource name to its network address, such systems do not require a priori knowledge of some unique identifier of the resource, like its network address, and hence can be used to dynamically discover and utilize resources as they become available in a pervasive system.

Such attribute-based resource discovery systems differ in the format used by them to describe resource properties, expressiveness offered by their look-up interfaces, whether they offer push-based or pull-based discovery and whether queries are mediated by a directory service or resolved in a peer-to-peer fashion in the system. In addition to the classical examples like Grapevine, GNS [19] and X.500 [20], a range of industrial standards like Microsoft's UPnP resource discovery protocol (SSDP) [11], IBM's T-Spaces, and IETF's Service Location Protocol [9], and experimental systems like MIT's INS [4] and Berkeley's SSDS [3] have emerged over the last few years. For example, where SLP offers a rich LDAP-based [20] query interface, systems like INS and SSDP define simple attribute-based resolvers that can be hosted in resource constrained mobile devices.

SoNS is designed such that different discovery protocols can be added to its resource discovery module, possibly via a simple wrapper function to covert the SoNS attribute list to the specific format used by a discovery protocol, e.g. XML (used by Berkeley's SSDS).

Unlike existing resource discovery protocols that simply match queries against resource descriptions, SoNS uses an applications-specific evaluation framework that continuously monitors, evaluates and compares the available alternatives in order to pick the closest match to application requirements. Indeed, the problem of satisfying high-level requirements with imperfect resources has been extensively researched in the AI domain [12]. However, where systems like MetaGlue [12] propose to use general-purpose constraint satisfaction engines over complex utility functions, SoNS default evaluator is designed to be simple and responsive to changes in the system.

Content-based routing systems like INS's late binding architecture [4] and Information Bus [21], as well as application-level anycast routing systems like [8], allow applications to send messages without specifying the network address of the recipient, and route the messages to the appropriate server by looking at the content of each network message and matching that with the properties of the available servers. Where such systems offer an alternative to our approach, they inherently lack application-level session semantics, do not offer a clean interface for configuring the application-specific policy for resource comparison and session-rebinding, introduce the overhead of resolving service descriptions within the critical path of every message delivery, and, by defining their own routing framework, do not leverage the support for QoS offered by the underlying network.

12 Conclusions and Future Work

This paper establishes the need for *service-oriented* network connection, and presents the design and implementation of the SoNS system. SoNS presents an application with an extended socket interface to open a service-oriented network connection by providing a high-level service-specification. When asked to establish a connection, SoNS discovers the available resources and connects the application to the resource that best provides that service in its context. Once connected, SoNS continuously monitors, compares, and evaluates available alternatives, and reconnects the application to a better alternative if one becomes available.

As opposed to content-routing systems, SoNS moves the cost of discovering and evaluating resources against application requirements at the connection set-up time, and allows the application to exercise control at the level of a network session. Since the cost of discovery and selection in SoNS is amortized over the life of the network session, it allows SoNS to be significantly more sophisticated in terms of expressiveness, evaluation and selection of available resources, as compared to systems that perform message-level service-selection-and-routing.

As SoNS integrates support for context-awareness with a traditional operating system communication interface, we have found it much simpler to use than other systems that require the use of additional, and often several different [7], APIs to build a context-aware application. Though we believe that SoNS has the potential to become an integral part of future operating systems in a pervasive computing environment, it relies on the wide-spread deployment of network devices embedded with service advertisement protocols, as well as the availability of location detecting mechanisms to estimate the distance of a user with the devices embedded in her context.

The design of SoNS pays special attention to extensibility in order to take advantage of the wide range of emerging technologies for resource discovery, location detection and network connection migration.

The current design of SoNS does not include a security framework. Security in such a system is required at several levels: to protect resources against illegitimate access, to protect the SoNS system against malicious extensions, and to protect the connection migration system against connection hijacking. Though some discovery protocols, like SSDS and SLP, and connection migration schemes, like Migrate, define their own security models, we are currently investigating an extensible security framework that would allow security policies to be defined independently of the constituent modules.

SoNS makes it simpler to develop context-aware applications. Our experience with SoNS has shown us that unlike message-based routing systems that are better suited to *command-based applications* e.g. "sending a document to the nearest printer", SoNS is equally useful for *connection-oriented applications* as well, e.g. follow-me-video/audio. We are currently developing more applications to demonstrate the utility of SoNS in pervasive mobile environments.

References

1. Christopher K. Hess et al. *Building Applications for Ubiquitous Computing Environments*, International Conference on Pervasive Computing (Pervasive 2002), pp. 16-29, Zurich, Switzerland, August 26-28, 2002.
2. Esler, M. et al. G. *Next Century Challenges: Data-Centric Networking for Invisible Computing: The Portolano Project at the University of Washington* Mobicom 99
3. S. Czerwinski et al. *An architecture for a secure service discovery service.* In Proc. of MobiCom-99, pages 24-35, N.Y., August 1999
4. William Adjie-Winoto et al. *The design and implementation of an intentional naming system*, Proc. 17th ACM SOSP, Kiawah Island, SC, Dec. 1999.
5. David Garlan et al. *Project Aura: Towards Distraction-Free Pervasive Computing* IEEE Pervasive Computing, special issue on "Integrated Pervasive Computing Environments", Volume 1, Number 2, April-June 2002, pages 22-31.
6. C. Perkins et al *A Mobile Networking System Based on Internet Protocol*, IEEE Personal Communications, Vol. 1, No. 1, pp. 32-41, March 1994.
7. Harter, A. et al.: *The anatomy of a context-aware application.*, Mobile Computing and Networking. (1999) 59-68
8. S. Bhattacharjee et al. *Application Layer Anycasting.* In Proc. IEEE INFOCOM'97, 1997
9. Erik Guttman. *Service Location Protocol: Automatic Discovery of IP Network Services.* IEEE Internet Computing Journal, 3(4), 1999.
10. J. Jaffar et al. *Constraint logic programming: A survey.* The Journal of Logic Programming, 19/20:503--582, May/July 1994.
11. *Universal Plug and Play*, http://www.upnp.org
12. Krzysztof Gajos. *Rascal - a Resource Manager for Multi Agent Systems in Smart spaces.* In Proceedings of CEEMAS 2001.
13. Alex C. Snoeren et al. *Fine-Grained Failover Using Connection Migration,* Proc. 3rd USENIX USITS, March 2001.
14. R. Golding. *A Weak-Consistency Architecture for Distributed Information Services.* Computing Systems, 5(4):379--405, 1992.
15. Alex C. Snoeren et al. *An End-to-End Approach to Host Mobility*, Proc. 6th ACM MobiCom, August 2000
16. R.R. Stewart, et al. *Stream Control Transmission Protocol.* RFC 2960, IETF, Oct. 2000.
17. Nissanka B. Priyantha et al. *The Cricket Compass for Context-Aware Mobile Applications* Proc. ACM MOBICOM Conf., Rome, Italy, July 2001.
18. Jon Salz, SM thesis, MIT. 2002, *The Transparent Extensible Session-Layer Architecture for End-to-End Network Services.*
19. A. Birrell et al. *Grapevine: An exercise in distributed computing.* Comm. Of the ACM, 25(4):260–274, April 1982.
20. *CCITT. The Directory—Overview of Concepts, Models and Services*, December 1988. X.500 series recommendations, Geneva, Switzerland.
21. B. Oki et al. The Information Bus (R) – *An Architecture for Extensible Distributed Systems.* In *Proc. ACM SOSP*, pages 58–78, 1993.
22. MIT Project Oxygen, http://www.oxygen.lcs.mit.edu

Acknowledgements

We would like to thank Steve Ward for his valuable comments that helped us improve the paper significantly. The title of the paper was also suggested by Steve Ward.
We would like to thank Marvin Theimer, our "shepherd", and anonymous reviewers for their helpful comments and suggestions.

Energy-Conserving Data Placement and Asynchronous Multicast in Wireless Sensor Networks

Sagnik Bhattacharya, Hyung Kim, Shashi Prabh, Tarek Abdelzaher
Department of Computer Science
University of Virginia
Charlottesville, VA 22904

Abstract

In recent years, large distributed sensor networks have emerged as a new fast-growing application domain for wireless computing. In this paper, we present a distributed application-layer service for data placement and asynchronous multicast whose purpose is power conservation. Since the dominant traffic in a sensor network is that of data retrieval, (i) caching mutable data at locations that minimize the sum of request and update traffic, and (ii) asynchronously multicasting updates from sensors to observers can significantly reduce the total number of packet transmissions in the network. Our simulation results show that our service subsequently reduces network energy consumption while maintaining the desired data consistency semantics.

1. Introduction

Sensor networks are ad hoc wireless networks made of large numbers of small, cheap devices with limited sensing, computation, actuation, and wireless communication capabilities. Such a network, for example, can be dropped from the sky on a disaster area to form collaborative teams of programmable nodes that help with rescue operations. Sensor networks are made possible by advances in processor, memory and radio communication technology, which enable low-cost mass-production of sensor-equipped wireless computing nodes.

The sensor network paradigm is motivated by applications such as guiding rescue efforts in disaster areas, monitoring poorly accessible or dangerous environments, collecting military intelligence, tracking wild-life, or protecting equipment and personnel in unfriendly terrains. In such environments, it is usually impractical to build fixed infrastructures of powerful and expensive nodes. Instead, the sensor networks philosophy advocates the use of myriads of inexpensive nodes strewn arbitrarily in the environment and left largely unattended.

The primary function of sensor networks is the collection and delivery of sensory data. Power is identified as one of the most expensive resources. Due to the difficulty of battery recharging of thousands of devices in the remote or hostile environment, maximizing battery lifetime by conserving power is a matter of great importance.

In this paper, we develop a distributed framework that improves power conservation by application-layer sensor data caching and asynchronous update multicast. The goal of the framework is to reduce the total power expended on the primary network function; namely, data collection and delivery.

The importance of optimizing communication cost is also supported by measured data from recent prototypes of sensor network devices, which show that the main power sink in the network is, indeed, wireless communication. For example, the Berkeley motes [15] consume 1 μJ for transmitting and 0.5 μJ for receiving a single bit, while the CPU can execute 208 cycles (roughly 100 instructions) with 0.8 μJ. Assuming full load, CPU power consumption is about 10mW, compared to 50mW for the radio. The high power cost of communication makes it a prime candidate for optimization.

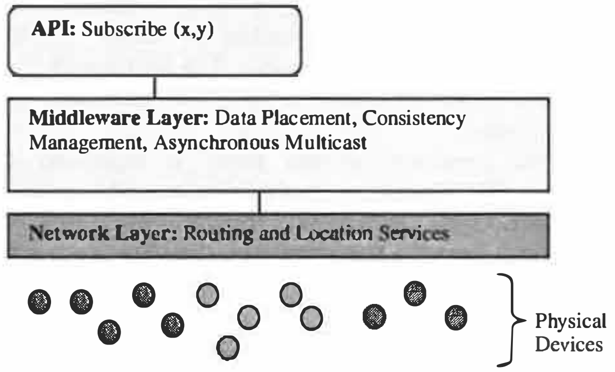
The remainder of this paper is organized as follows. Section 2 presents the service model and the formulation of the power minimization problem. Section 3 presents the details of the data placement middleware and its API. Section 4 presents an evaluation using experimental as well as simulation results. Section 5 reviews the related work. The paper concludes with section 6.

2. Service Model

Consider a dense *ad hoc* wireless sensor network with multiple observers, spread over a large monitored area. At any given time, the observers' attention is directed to a relatively limited number of key locales in the network, where important events or activities are taking place. We call them *focus locales.* For example, in a disaster area scenario, rescue team members may be interested in monitoring survivors. The locations of found survivors therefore represent the focus locales of this application. The total number of sensor nodes is assumed to be much larger than the number of focus locales at any given time.

Sensor nodes at each focus locale elect a local representative for communication with the rest of the world. Distributed leader election algorithms may be borrowed for this purpose from previous literature and are not the goal of this paper. Our service adopts a publish-subscribe model, as shown in Figure 1. In this model, each representative publishes sensory data about its focus locale to observers who subscribe to a corresponding multicast group to receive such data. The size of the published update stream originating at a given locale is time-varying, depending on the volatility of the environment and the type of sensors involved. An environment, which changes frequently, will generate more update traffic than a quiescent environment. Similarly, sound sensors (microphones) will generate more traffic than temperature sensors.

Contrary to previous multicast frameworks for sensor networks, update traffic is multicast from focus locales to receivers in an *asynchronous* manner. Data caches are created at the nodes of the multicast tree. A lazy algorithm is used for propagating data updates among neighboring caches along the tree in the direction of the receivers. These receivers may be wireless hand-held devices or laptops, for example, in the possession of rescue team members operating in a disaster area. We assume that receivers do not move, or move slowly compared to communication delays in the network.



API: Subscribe (x,y)
Middleware Layer: Data Placement, Consistency Management, Asynchronous Multicast
Network Layer: Routing and Location Services
Physical Devices

Fig. 1: Middleware Architecture

In general, data updates can be either *accumulative* or *non-accumulative*. An example of accumulative updates is recorded sound. To receive a continuous recording, *all* (or *most*) sound samples should be communicated. An example of non-accumulative updates is thermal measurements. If the application is interested in the current temperature only, past temperature updates need not be reported. Most real-time sensor outputs, with the general exception of multimedia data, are non-accumulative in that current measurements subsume stale measurements. Hence, our scheme is restricted to non-accumulative updates. This decision is also motivated by the fact that current sensor network technology is too slow to handle multimedia traffic in a cost-efficient way.

While in this paper we do not consider streaming multimedia, an argument in favor of addressing such traffic in sensor networks is that more powerful devices may become available in the foreseeable future at an affordable price. We argue, however, that advances in sensor network technology are most likely occur in *two* directions: developing more powerful devices of the same form factor, and developing smaller devices of the same processing and communication capacity. Research reported in this paper is more relevant to the latter direction.

In our model, observers who join the asynchronous multicast tree specify a period at which the requested data should be reported. Flurries of changes in the environment need not be individually reported if they occur at time-scales smaller than this period. Different observers may specify different period requirements for the same measurement. For example, an observer who is close to the measured activity may request a higher reporting rate than a distant observer.

Our middleware achieves four main functions; (i) it determines the number of data caches for each focus locale, (ii) it chooses the best location for each cache such that communication energy is minimized, (iii) it maintains each cache consistent with its data source at the corresponding focus locale, and (iv) it feeds data to observers from the most suitable cache instead of the original sources.

A key difference between this problem and the problem of caching in an Internet context is that in the latter case, the topology of the network restricts the choice of cache locations. In contrast, we assume a sensor network that is dense enough such that a data cache can be placed at any arbitrary physical location in the monitored region, offering new degrees of freedom to the data placement algorithm. Another key difference is that the number of Internet proxy caches is typically much smaller than the number of different web sites. Hence, such caches are centralized powerful machines, which gather and retain content from a large number of distributed sources. In contrast, in this paper, we consider a middleware caching service, which runs on *every* sensor node. Since the number of sensor nodes is larger than the number of focus locales, the storage requirements of this service on any single node are very small.

We assume that sensor nodes know their location. Algorithms for estimating geographic or logical coordinates have been explored at length in the sensor network research [5][6]. These efforts address the problem of location awareness using algorithms that do not require high cost devices such as GPS on every node. Classical ad hoc wireless routing protocols like AODV [8], and DSDV [9] may be used along each unicast edge of our data dissemination tree. These protocols, however, are not location-aware which may affect performance. Several more recent adaptations such as Location-aware routing (LAR) [7] and geographical forwarding [4] make use of the location information. These routing algorithms would be a natural choice for the network layer underneath our service. We now formulate our data placement problem mathematically.

2.1. Problem formulation

Consider a sensor network that is monitoring a set of focus locales at which events of interest occur. Given a locale *(X,Y)* in a sensor network, let BS= $\{BS_1, BS_2, \ldots, BS_M\}$ be a set of *M* observers that request data from that locale with rates $R_{req}=\{R_1, R_2, \ldots, R_M\}$. Let sensor updates at *(X,Y)* occur at an *average* rate R_{update}. A tree of copies is created for the sensor as shown in Figure 2.

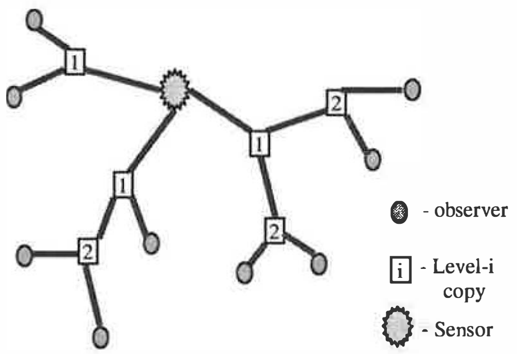


Fig 2: Creating a hierarchy of copies

We define the cost of message transfer between two nodes in the tree as the power expended on a packet's transfer on the shortest route multiplied by the packet rate. Consider the case of placing a single data copy to minimize cost as defined above. Let the data copy be placed at a distance n_i hops from the i^{th} observer and at a distance n_{sens} hops from the sensor node serving the data. In a densely populated network, the hop counts will be large. The cost of sending a single packet is proportional to the hop count. Hence, the net cost of serving all observers is:

$$T = n_{sens} \cdot R_{update} + \sum_{1 \le i \le M} n_i \cdot R_i \qquad (1)$$

To place the copy at the optimal location, *T* has to be minimized. Figure 3 shows the situation with three observers. We can reduce this problem to the following geometric optimization. Given *N* points, where point *i* is at location (X_i, Y_i), find a point *(x,y)* such that $D = \sum_{1 \le i \le N} (d_i \cdot w_i)$ is minimum, where, d_i is the distance of the i^{th} point from *(x,y)*, and w_i is the weight of the edge from the i^{th} point to *(x,y)*. This is illustrated in Figure 4. A heuristic solution to this problem is to place *(x,y)* at the center of gravity of the *N* input points in question, i.e.:

$$x = \sum_{1 \le i \le N} x_i w_i / \sum_{1 \le i \le N} w_i \qquad (2)$$

$$y = \sum_{1 \le i \le N} y_i w_i / \sum_{1 \le i \le N} w_i \qquad (3)$$

Hence, in a minimum-cost tree with multiple copies (i.e., multiple internal vertices), each copy *(x,y)* should be at the center of gravity of those vertices to which it is connected. The objective of our algorithm is to find such a tree.

In the following, we compare our formulation to other popular variants of content placement problems described in prior literature. If the number of copies in the tree is known in advance, a popular variation of the problem is expressed as a Minimum *K*-median problem, stated as follows. Given n points (possible copy locations), select *K* of them to host data copies, and feed each observer from a copy such that total communication cost *D* is minimized, where:

$$D = \sum_{1 \le j \le K} \sum_{1 \le i \le N} c_{ij} \cdot y_{ij} \qquad (4)$$

c_{ij} is the cost of the edge from *i* to *j* and y_{ij} is 1 if the j^{th} copy serves the i^{th} observer, and 0 otherwise. Many Internet-based content placement algorithms adopt this model. In this case, the possible locations of the caches are fixed. Hence, c_{ij} is fixed for the given network topology. The problem is NP-hard, but heuristic solutions are possible, e.g., [10] and [11]. If the cache locations are specified, a minimum spanning tree can be constructed to disseminate information from senders to receivers at the lowest cost.

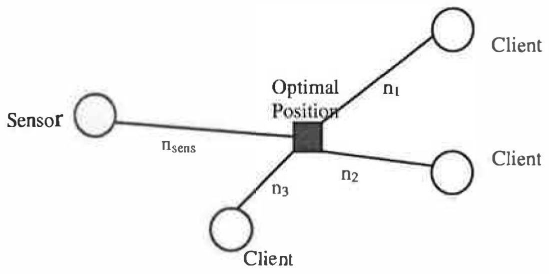


Fig. 3: Scenario for M = 4. The three base-stations are served by the same copy.

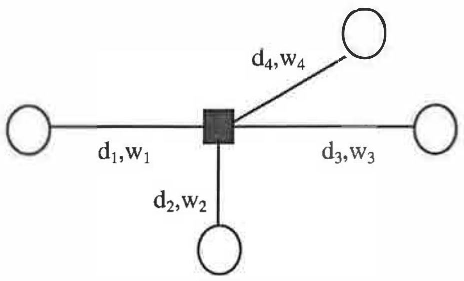


Fig. 4: Corresponding geometric problem for N=4.

Our model differs in that copy locations are not known *a priori*. In a dense sensor network, the number of nodes n approaches infinity. Copies can essentially be placed anywhere in the Euclidean plane without restrictions. In this case, the problem is that of constructing a minimum-cost weighted Steiner tree, which connects the sensor node to the observers.

The Steiner tree formulation differs from the K-median and spanning tree problems in that it allows one to create new nodes in the tree as opposed to having to choose from a pre-specified set of possible node locations. This difference separates our paper from similar work in web caching and content distribution literature.

Note that R_{update} in our algorithm is not a fixed sampling rate, but rather refers to the average rate of change of the environment. Hence, it may vary dynamically with environmental conditions. For example, it may decrease when the environment is quiet. An advantage of such dynamic adaptation is that no energy is wasted when no updates occur. A disadvantage is that an application is unable to tell when it has missed an update (e.g., due to message loss), since it does not expect updates to arrive at particular time intervals. This problem can be solved in several ways.

First, we may let R_{update} be a fixed sampling rate. The formulation of our algorithm remains the same. In this case, if a sample does not arrive in time, the application can tell. Alternatively, the origin sensor may number the updates. If a gap occurs in the received update numbers, the application is aware that a previous update was lost. The occasional loss may be acceptable since we assume that only the latest update is relevant at any given time. A potential problem with the latter approach is that in the absence of subsequent environmental changes, an important update may be lost, unbeknown to the application, indefinitely. One solution is to enforce an upper bound, *B*, on the update period. Hence, when the environment is quiet a message is expected at least once every *B* seconds. Otherwise, the application is aware of a problem. In the rest of the paper, we shall not address the issue update loss any further.

3. Data Placement

Upon perturbation, distributed physical systems such as weights interconnected by strings settle into an equilibrium position, which represents a minimum energy state. Our data placement algorithm is inspired by such systems. Assuming environmental conditions don't change, each step of the algorithm reduces a measure of total energy until a minimum energy tree is found. More specifically, we use a distributed greedy heuristic that iteratively places each node at the center of gravity of its neighbors. Note that, while in a physical system, energy has a direct meaning, in our system energy is an abstract mathematical quantity. We call the depth of the copy in the distribution tree rooted at the origin sensor, the *copy level*. The original data at the sensor is referred to as the level-*0* copy. A heuristic is used to add or remove copies in the tree. The algorithm is described in more detail next.

3.1 The Algorithm

Each node on the multicast tree rooted at the sensor maintains a location pointer to its parent as well as a location pointer to each of its children. One can think of these pointers as an application-layer routing table. For each child, the node maintains the *maximum propagation rate*, which is the maximum of all requested update rates of all observers served by that child. A node never forwards updates to a child at a rate higher than the child's maximum propagation rate. This way, flurries of environmental updates that exceed some receivers' requested rates are not propagated unnecessarily to those receivers.

3.1.1 Joining the Multicast Tree

An observer, *k*, joins a multicast tree by sending a join() message to the location of the origin sensor, i.e., to the level-*0* copy. The message indicates the location of the observer and its desired update rate R_k. The origin sensor forwards the message along the multicast tree in the direction of the new observer as follows. Each level-*i* copy (starting with the origin sensor), upon receipt of the join message, determines if the new observer is closer to any of its children than to itself. If so, it forwards the join message to the corresponding child, i.e., to a level-*(i+1)* copy. If the maximum propagation rate for that child is lower than R_k it is changed to R_k. This recursive forwarding terminates when a node is found with no children that are closer to the observer. We call this copy the *nearest neighbor*. The nearest neighbor adds the observer to the set of its children. The maximum propagation rate for the observer is initialized to its requested update rate. Figure 5 illustrates the message exchange in the join process.

3.1.2 Copy Creation and Migration

For the purposes of creation of new cache copies, nodes are differentiated into fixed and migratory. The origin sensor and observers are fixed nodes. Other nodes are migratory nodes that can move to better locations of fork off new copies.

When a newly joined observer is connected to its nearest neighbor *N*, node *N* computes the center of gravity of itself and all its neighbors. It then computes the savings, if any, resulting from creating a new copy at that center of gravity. If the savings from creating the copy exceed a threshold, the option of creating this copy is deemed *viable*. Before we proceed further, let us look more closely at how the copy may be created.

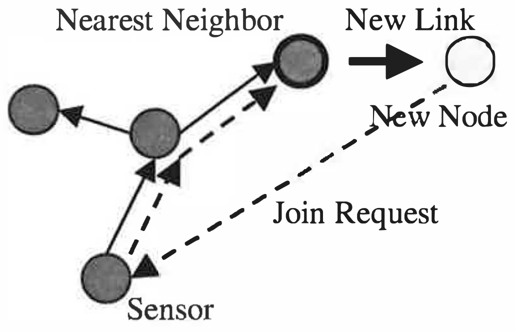


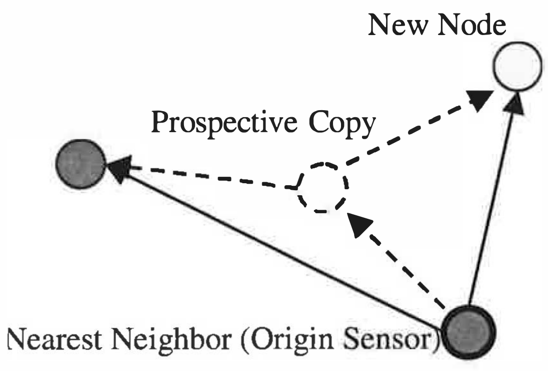
Fig. 5: Joining the Multicast Tree

If *N* (the nearest neighbor) is the origin sensor, the new copy can only be created *downstream* from it. The copy would be fed from *N* and in turn feed *N*'s children as shown in Figure 6-a. Otherwise, if *N* is not the origin sensor, the new copy can in principle be created either downstream or upstream from *N*. An *upstream* copy would be fed from *N*'s parent and would feed both *N* and *N*'s children as shown in Figure 6-b. A downstream copy would be created as described above (Figure 6-a). Observe that, if *N* is not a fixed copy, a third option is also possible. Namely, it is possible to simply move *N* to a new position. This is called copy *migration*. In copy migration, when a newly joined observer is connected to a migratory nearest neighbor *N*, the node computes the center of gravity of all its neighbors (including the new observer), and evaluates the savings that would arise if it moves to the computed position. If the difference is larger than a fixed threshold the option of migration is deemed *viable*. This is illustrated in Figure 6-c.

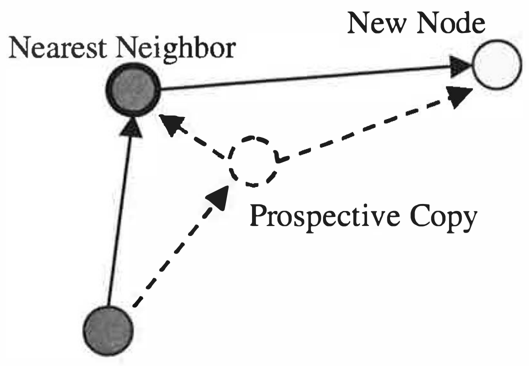
A viable option with the maximum savings among three data placement options described above is executed. It is easy to show that no new copies are created unless there are three nodes in the system, and that at most one copy is created for every newly joined member. Hence, the algorithm creates at most *m-2* copies where *m* is the total number of observers.

3.1.3 Leaving the Multicast Tree

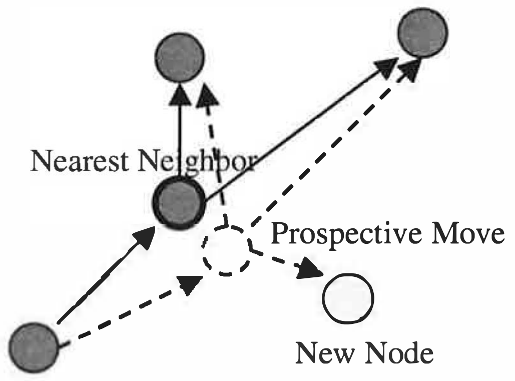
An observer, *k*, leaves the multicast tree by sending a leave() message to its parent *N*. The parent stops forwarding messages to the departed observer. If *k* had the highest maximum forwarding rate among *N*'s children, *N* resets its own maximum forwarding rate to that of the next-highest rate child. If *N* is a migratory node, it computes the center of gravity of all remaining neighbors, computes the savings that result from moving to that center, and moves there if the savings exceed a threshold. If there is only one child left for the migratory node, the node is deleted and its parent takes over its child.



(a) Nearest neighbor creates downstream copy



(b) Nearest neighbor creates upstream copy



(c) Nearest neighbor moves

Fig. 6: Copy Creation and Migration Rules

3.2 Sampling R_{update}

To perform center of gravity computations, nodes must know not only the requested observer rates, but also the environmental sensor update rate, R_{update}. There are two simple approaches towards the measurement of that rate. One approach is to measure the number of updates over the last n seconds. A disadvantage of this approach is that it has a fixed time horizon after which it forgets the past. It may be more advantages to adapt the horizon to the current rate of updates itself, such that system agility is increased when activity is high. An approach for calculating R_{update}, which has the aforementioned adaptive property, is to take the inverse of the average of the last k inter-arrival times. More complicated methods like predictive modeling could have been used but this would be limited by the computation and storage resource constraints of the nodes. For our simulation purposes we have used a simple model where R_{update} is calculated as the inverse of the average of the last five inter-arrival times. The number five is selected as the sample size to reflect that we expect any five consecutive updates to be strongly correlated, though a larger or smaller number could be chosen according to how volatile we expect the environment to be.

4. Evaluation

Our current service implementation utilized Berkeley motes [15] as the underlying distributed platform. These are tiny computing devices, which run a microthreaded operating system called TinyOS [16]. Each node has up to three sensors. It runs on an 8-bit 4 MHz micro-controller and has 128KB of program memory and 4KB of data memory. However, the number of motes available to us at present is insufficient for large-scale experiments. Hence, in the first set of experiments, we use the motes only to derive communication and power consumption characteristics that are then fed to a simulator. Accordingly, we also implemented our data placement middleware in the ns-2 simulator [20]. Our goal in simulating the data placement algorithm is to test whether it actually conserves power as expected given the power and communication characteristics of the motes. Our simulation model is a network of ($200 \leq N \leq 500$) nodes in a 200m x 200m grid each having a radio range of 20m. The packet sizes are kept as 30 bytes, as used by the Berkeley motes running TinyOS. The number of base-stations is roughly 5% the number of nodes in the network. Since we have to model a location-aware network, we assume that each node knows it own location. We implemented a wrapper, which works on top of the simulated routing protocol in ns-2, and translates the destination location of packets into actual destination node addresses. A focus locale in the network is generated at random for simulation purposes, and the base-stations send periodic queries to the hot spot. The request rates are generated at random with a specific average throughout the experiment. The energy consumption is measured in terms of Joules per node per flow. Each value is taken as the average over three simulations.

At peak load the Berkeley motes consume about 60mW of power [15]. Of this 60mW about 10mW is consumed by the microcontroller unit. However, the MCU is not loaded about 50% of the time, and it has an idle mode in which it consumes only 40% of the normal power. Another very important fact to note is that more than 90% of the CPU utilization is due to bit, byte and packet level processing. Thus reduction in the number of packets in the network leads to lower CPU utilization and hence even greater power savings. However for simulation purposes, we have no accurate way to model CPU utilization in ns-2, so we measure only the radio power consumption, and the energy spent in communication. Since we discount savings in CPU power consumption, the actual power savings of the algorithm may be higher than shown in this section. The importance of optimizing communication cost is also supported by measured data from recent prototypes of sensor network devices, which show that the main power sink in the network is, indeed, wireless communication. Energy consumption for communication in our simulation follows the Berkeley motes [15] which consume 1 μJ for transmitting and 0.5 μJ for receiving a single bit. We also use the two-ray ground model ($1/r^4$) as the radio propagation model and an omni-directional antenna having unity gain in the simulation.

Geographic forwarding (GF) [29] is found as a routing protocol appropriate for the sensor networks. It is therefore used in the simulation. GF makes a greedy decision to forward a packet to a neighbor if it has the shortest geographic distance to the destination among all neighbors and it is closer to the destination than the forwarding node. To illustrate the appropriateness of GF, we compare its performance to that of AODV whn they are used in conjunction with data placement. AODV is a reactive routing algorithm developed for ad hoc wireless networks. It computes and caches routes on-demand. As shown in Fig. 7, the power consumption measured for GF is lower than that for AODV. The primary reason is that AODV does not leverage geographical information, thereby consuming more energy on route discovery. Hence for our simulations, we use GF as a routing algorithm.

4.1 Simulation Results

We compare the performance of the data placement middleware against four baselines; (i) a simple unicast-based query-response model, (ii) update multicast (synchronous push model) (iii) directed diffusion, and (iv) update flooding.

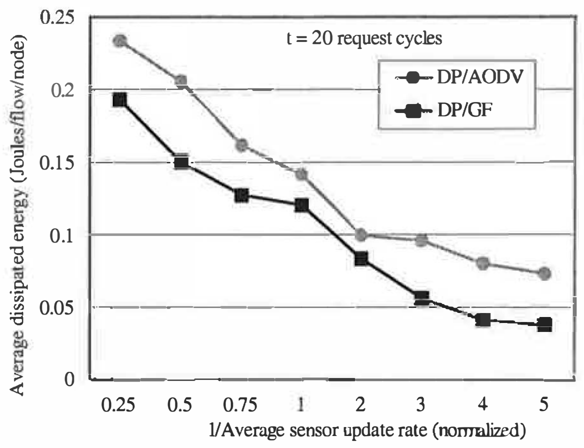


Fig 7: Average dissipated energy of data placement over AODV and GF

The first experiment (Figure 8) compares the energy consumption of the four aforementioned baselines for different node densities. For this experiment, the request rate is set, on average, to about two times the average update rate of the environment. Hence, regular (as opposed to asynchronous) multicast should be an optimal policy. We vary the number of nodes in the network over a grid of 200m x 200m. As we can see from Figure 8, flooding performs very badly. Traces reveal that power is wasted on both excessive communication and collisions caused by the update messages flooding through the network. The query-response scenario performs much better than flooding, but not as good as data placement. Directed diffusion performs almost as good as data placement. As expected, regular multicast performs best. The slight difference between regular multicast and data placement is due to the somewhat higher overhead of our scheme. As we show shortly, this overhead is offset by considerable savings when the average update rate increases beyond the request rate.

Unlike regular multicast, a main feature of the data placement algorithm is that it is adaptive and sensitive to changes in average sensor update rates. When the sensor update rates are high, more replicas are refreshed at a rate determined by the request rates and when the update rate is low, the copies are refreshed only when an update actually occurs. Next we compare the performance of the four baselines to that of our adaptive heuristic as the average update rate is changed. These communication models are compared over a network of 400 nodes in a 200m x 200m grid each having a radio range of 25m. The sensor update rate is normalized against the average request rate. The sensor update period is varied between 1.5 s and 30 s. The average request rate is one per six seconds.

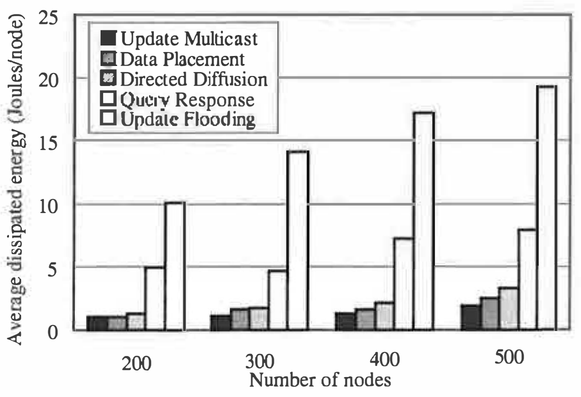


Fig 8: Average dissipated energy

In Figure 9, we show the average energy consumption in the steady state after all observers have joined the tree.

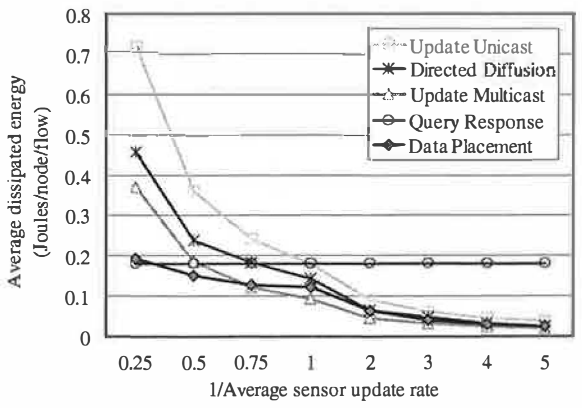


Fig 9: Steady-state dissipated energy plotted against sensor update rate.

As can be seen from the figure, when the Sensor Update Rate is high (i.e., 1/ Sensor Update Rate <1), the data placement algorithm performs better than the simple query response model, the directed diffusion, and update unicast because it does not send unnecessary updates. When the network is more quiescent, it achieves results, which are quite close to the simple multicast strategy. The difference between the two is the power overhead of adaptation. Thus, our data placement middleware adapts to the volatility of the environment resulting in a performance that is better than update multicast when the update rate is low, but does not suffer a performance degradation in when the sensor update rate is high.

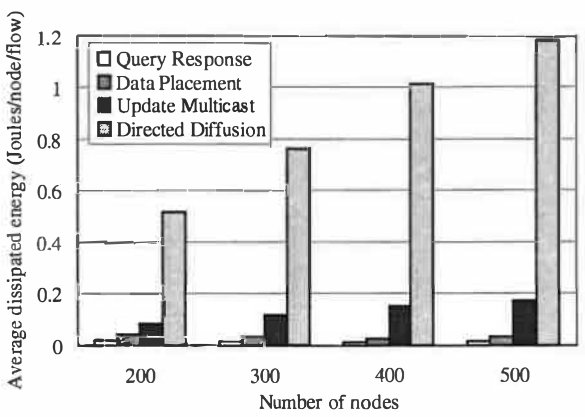


Fig 10: Energy consumption in transient phase

The next experiment is carried out in order to measure energy consumed when a new observer joins the tree. In the data placement, the new observer sends a join message to the origin sensor and the nearest node is found to connect the observer. Figure 10 shows the average energy consumption per join during the transient phase of tree construction. It is evaluated under the same environmental conditions as in the preceding experiment. The transient phase in the case of directed diffusion includes both initial flooding and multi-path sending before a single path is chosen by the gradient. The data placement heuristic consumes less energy than the update multicast and the energy dissipation does not depend upon the number of nodes in the network.

In our data placement algorithm, when a new data copy is placed, neighbors of the new copy are no longer in the center of gravity of their own neighboring nodes. Hence, as a rule, when the number or locations of neighbors of some data copy change, the copy recalculates its position to the new center of gravity. If the difference between the cost at the new position and the cost at the old one is larger than a given threshold, the old copy migrates to the new position. This migration, in turn, may cause other surrounding nodes to migrate. The effect percolates through the tree branch until no more nodes need to be moved. Figure 11 shows a transient and steady state energy consumption graph against various threshold values in the data placement heuristic. A smaller threshold results in more aggressive tree optimization. It entails more overhead for tree construction, but results in lower power consumption at steady state. Conversely, an extremely large threshold makes the tree essentially more static. Transient overhead is reduced, but steady state power consumption increases due to a less optimized tree. Observe that the difference between power consumption at the two extremes is not that pronounced. This is because network density is not infinite. Hence, a newly computed copy location is likely to fall in a void between sensor nodes. As an approximation, the copy is mapped to the closest node from that void. Even if the threshold is zero, it is often the case that the new (optimized) position of a copy is close enough to its current position so that the copy is mapped to the same node as before. No move is therefore taken making the performance more similar to that of a static tree.

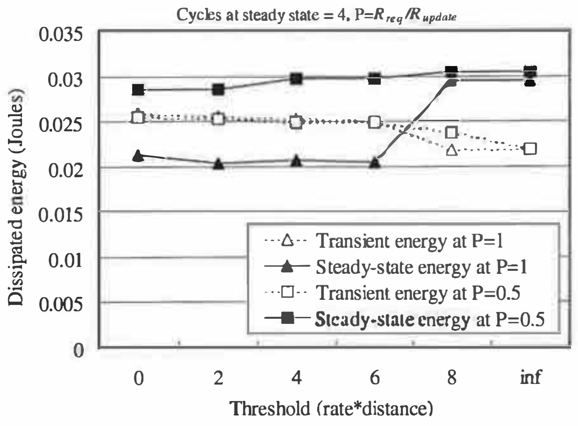
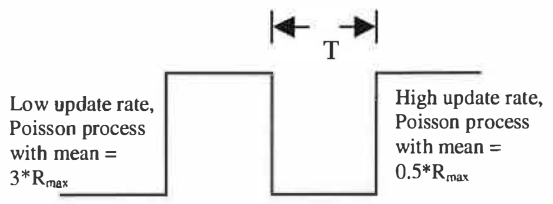


Fig 11: Energy consumption in adjusting data placement

Next, we test system performance when the average update rate is a non-stationary process, which changes drastically over short periods of time. We use the average update traffic pattern shown in Figure 12. The sensor switches between high and low average update rates with a period T. This average rate is used to determine actual update inter-arrival times, drawn from a Poisson distribution. This load model is used to simulate quiescent periods in the environment interlaced with volatile periods. At high update times, the update rate is made to be a Poisson distribution with mean of 0.5 times the maximum request rate, and at low update times it is made to be a Poisson distribution with mean of 3 times the request rate.



T
Low update rate, Poisson process with mean = $3*R_{max}$
High update rate, Poisson process with mean = $0.5*R_{max}$

Fig 12: Update pattern

Figure 13 shows the performance of the data placement algorithm for different values of T. The network size was fixed at 300 nodes over a 200m x 200m grid. When T is sufficiently larger than R_{max} (R_{max} is the maximum request rate), data placement performs better than both query-response and update multicast, because it adapts to suit both the high and low update rates. As the average update rate is calculated as the inverse of the average of the last five inter-arrival times, when T becomes less than $5R_{max}$, this sampling becomes increasingly inaccurate. Since the average update rate keeps changing rapidly, there are more failures and hence, the communication savings decrease as might be expected.

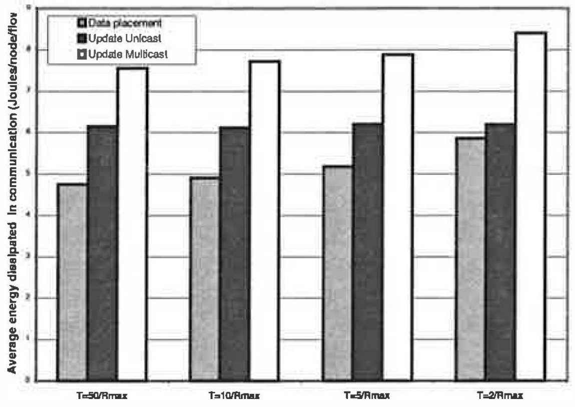


Fig 13: Average dissipated energy for the three models for different values of T

Another important concern to address is how the energy savings translate into an increase in the lifetime of the network. Figure 14 shows the fraction of nodes that are alive versus network run-time. We simulate a network of 400 nodes, keeping other parameters the same as earlier. Observers and the origin sensor are set to have infinite energy. A point to be noted here is that not all the nodes consume energy equally as some nodes are more active than others. When a certain number of nodes die, the network becomes partitioned, and parts of the network may become unreachable. At this point we consider the network unusable. Figure 14 shows the time it takes the network to get partitioned under various communication schemes. (The end of the plot indicates when the network gets partitioned.) It is shown that the system lifetime with the data placement heuristic is longer than with other baselines. As shown in Figure 9 and Figure 10, the data placement heuristic results in fair energy savings in both the steady state and the transient state. Hence, it increases system lifetime.

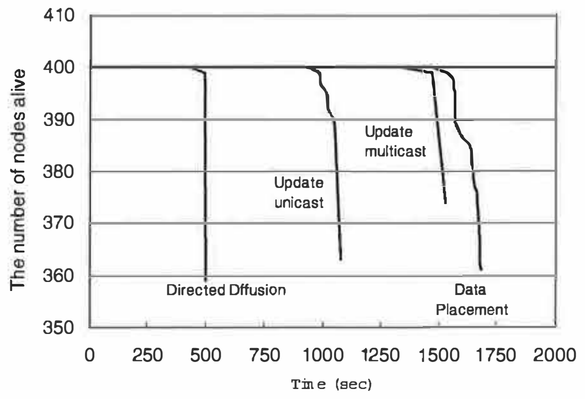


Fig 14: Lifetime of nodes in a sensor network using data placement

4.2 Effect of Sampling of R_{update}

As mentioned in section 3.3, there are several ways for accurately measuring R_{update}. We argue that inaccuracy in measuring/predicting R_{update} has very little impact on the overall performance of Data Placement. Figure 15 evaluates the sensitivity of power savings to the choice of the averaging interval for the sensor reported rate, R_{update}. Averaging intervals 5 and 10 are compared. The simulated traffic is the same as in Figure 12 and the experimental setup is the same as the experiment of Figure 13. The figure shows that power savings are insensitive to the averaging interval.

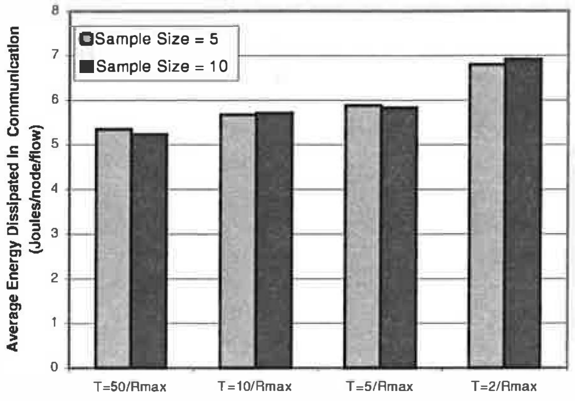


Fig 15: Average Dissipated Energy using different sample sizes for R_{undate}

Thus, our simulation results show that our data placement middleware gives us considerable energy savings irrespective of the amount of load or the dynamic nature of the network. A point to note is that the energy savings are not uniform over the whole network. They depend on the location of the base-stations and the locations from where data is requested, just as the communication without data placement would have expended energy non-uniformly. In general, we expect the results to improve with network size. This is because our main savings come from the use of adaptively constructed multicast trees. Since the size of a well-balanced tree is logarithmic in the number of recipients, power savings compared to unicast should increase exponentially with tree size.

4.3 Prototypical Testbed Implementation

To conclude our results, we constructed an experimental prototypical mini-testbed from a 5 by 5 grid of motes with one light sensor at a corner, and three base-stations (Figure 16). We ensured that a node can hear its immediately adjacent and diagonal neighbors. The three base-stations request light data from the sensor node, as shown in Figure 16. The base-station at the bottom right corner is connected to the serial port of a PC, where an application package reads the packets being received at the base-station.

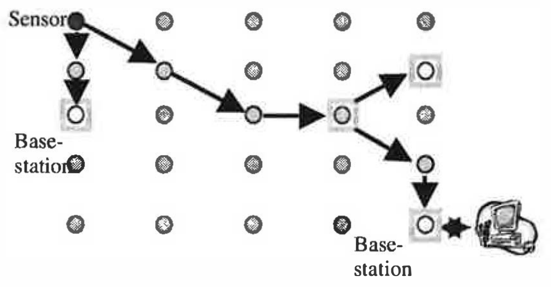


Fig 16: Sensor Network Testbed

The underlying routing protocol used is geographic forwarding, in which a node forwards a packet to the neighbor that is closest to its destination. The interface to the routing protocol accepts TinyOS commands from the data placement middleware to send messages to a given node. In addition, it signals an event when the message transmission is complete. The data placement layer handles this event.

The base-station (0) communicates with the PC using the serial port. Thus this base-station has both a RS232 communication channel as well as RF communication. In order to distinguish between the two channels, a special address (0x7e) is assigned to the serial port interface. Thus a device receiving a packet for this destination forwards the packet to the local UART instead of the radio.

The light sensor acquires 10-bit values for light intensities. The sensor data is acquired by polling the light sensor. We define a resolution range of 5 bits, i.e., an update occurs only if any of the 5 most significant bits change. We calculate the value of R_{update} by sampling the last 5 inter-update times.

Our implementation of data placement has about 250 lines of code (C statements), and the complete package including TinyOS and geographic forwarding takes about 14.5KB of program memory. Thus memory-wise, the middleware is not heavyweight in terms of memory.

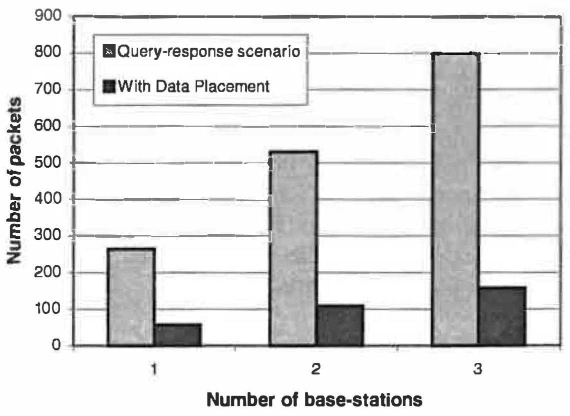


Fig 17: Number of packets using the two approaches

As has been discussed earlier, communication is the main sink for energy. The motes have an idle mode in which the mote draws 5μA compared to 5mA in the peak model. To estimate actual energy consumption on the motes, we measure the amount of communication in terms of the number of packets in the network. Figure 17 shows the number of packets in the network over a period of 200 seconds in a single trial run using data placement and without data placement. Here the sensor gets updated approximately once every 12 seconds. The base-stations request data once every 5 seconds. For the cases on one and two base-stations, all possible combinations of the three base-stations are considered, and the mean is taken.

Though the savings in the number of packets are an order of magnitude, this does not translate into proportional energy savings, since some power is expended on listening (even when nothing is being received). Even so, we have shown that we can get significant reduction in the number of packets in the network.

6. Related Work

Wireless sensor networks are a relatively new area of research. Traditional networking paradigms are not directly applicable to this scenario. However, there has been a lot of work lately on developing new paradigms and services for sensor networks, taking into account the unique features of these networks. New protocols are being developed for routing, MAC, data dissemination and location services. Several hardware platforms as well as specialized operating systems have also been developed. Since there are a number of parallel efforts, several different paradigms and protocols are bound to come up. Our data placement algorithm makes minimal assumptions about the underlying layers and the other supporting services.

One of the essential properties of a sensor network is that all the nodes are location aware. Since the nodes cannot afford to have heavyweight GPS, they have to use some location service, that estimates the location of the individual nodes using some GPS equipped nodes as beacons. Nagpal [5] and Bulusu, Heidemann et al. [6][12] proposed location estimation using beacons which does not require GPS at every node.

In a network with thousands of nodes, it would be wasteful to assign unique id's to each node. Also, queries would be addressed by location, and that would necessitate a location directory service, if unique id's were used. This would consume more resources. Papers by Heidemann, Silva et al. [3] and Imielinski, Goel [13] proposed addressing by geographical location and attributes. Our data placement algorithm shall work both for fixed addresses and addressing by location. For the case in which fixed addresses are used, a location directory service is used to lookup mapping from node-id's to locations.

To conserve energy from communication, Xu, Heidemann, and Estrin [4] have proposed a geographical adaptive fidelity (GAF) algorithm in which equivalence classes of nodes are formed from a routing perspective. Their method of energy conservation is to put nodes to sleep whenever possible. Our data placement algorithm can co-exist with GAF, by putting the constraint that nodes that are holding copies of the data cannot go to sleep until they handoff their data to another node.

The formation of a tree of copies along which the update is propagated is similar to the formation of a multicast tree, where all the nodes are members of the multicast group. In that sense, our work is related to multicast protocols. However, our approach differs from traditional multicast routing in several respects. First, updates are propagated asynchronously in a lazy manner in accordance with consistency constraints. Second, the depth of our tree is determined by the update and the request rates, and it adapts itself to minimize the communication. Finally, our work is an overlay multicast algorithm that works on top of the network layer, rather than traditional multicast routing [23][24] that takes place at the network layer. In wired networks, End-to-end multicast and Scribe based on Pastry, are included into overlay multicast designs.

Data placement is also similar to some of the ideas used in the placement of web server replicas [1]. In these schemes, replicas of web server content are placed online based on predicted demand. Data placement furthers this idea by using the property of location-awareness of the sensor nodes. Another approach used in [21] is to let the documents themselves decide their replication strategy. However, in our case we make use of the homogeneity of the data, and the fact that the sensors do not differentiate between types of data, to provide for a unified replication strategy. Another approach has been geographical push-caching [22] in which the server decides when and where to cache the files. This technique would not work for a sensor network because a sensor node cannot keep track of all the base-stations it is serving. In our approach, the sensor node only needs to keep track of a small number of level-1 copies.

ShopParent algorithm [30] is the latest work of publish/subscribe tree construction in the wireless adhoc network. This greedy algorithm builds the tree in a distributed fashion and uses a spanning tree to find a better outcome. Its model uses every node in the network for tree construction, which makes searching for the nearest node easier. However, the model is not available in the sensor network with a large number of nodes. The ShopParent is also different from ours in that it does not use a Steiner tree for multicast and does not use location information.

In recent years, research on Content Distribution Networks has focused on replication and placement of content to improve performance over a large scale distributed systems. Most of the work focuses on internet-type topologies and scale [26][25]. Chord [26] uses a variant of consistent hashing to map a key to a node. Its scalability lies in the fact that the amount of state that needs to be maintained by each node scales logarithmically with the number of Chord Nodes. In [27], the concept of Content Addressable Networks (CAN) is used for providing hashing like functionality to retrieve data from replicas in the network. CAN routing uses a co-ordinate routing table, and the network is visualized as a *d*-dimensional space. It extends functionalities of DNS by providing a flexible naming scheme. However replication in CAN is more hotspot driven. Another system, OceanStore [25], provides an infrastructure designed to span the globe and provide access to persistent information. The primary difference between these systems and data placement and replication in a sensor network is that the data placement algorithm presented in this paper leverages the location information available to the nodes to reduce power consumption. In contrast, work on CDN falls into two categories. It either (i) ignores physical topology altogether, focusing on peer-to-peer protocols defined in a logical overlay space, or (ii) optimizes the weight of the tree assuming a heterogeneous network of known topology with point-to-point links of different bandwidth. This optimization is not applicable to wireless sensor networks, where the physical network topology is unknown, yet it is the physical (i.e., geographic not overlay) tree that needs to be optimized for power consumption. Geographic information and energy consumption have been not considered in aforementioned papers.

Finally, the problem addressed in this paper is somewhat reminiscent of data placement in distributed shared memory systems [20]. As in shared memory systems, data in sensor networks can be thought of as a set of objects manipulated by *read()* and *write()* operations. The *write()* operations are performed by sensors. The *read()* operations are performed by users. In shared memory systems, it is desired to maximize the average performance of memory accesses given some desired data consistency semantics. In sensor networks, the problem is to minimize average power consumption per data access subject to data consistency constraints. Both problems reduce to finding algorithms for appropriate dynamic placement of data objects in the network such that communication is minimized. Sensor networks, however, due to their fine granularity, large scale, and direct interaction with the physical environment, exhibit significantly different data access patterns, consistency constraints, and communication cost models than do distributed shared-memory systems. Hence a new set of algorithms is called for to achieve power minimization.

7. Conclusion

In this paper, we have presented data placement as a means of reducing energy consumption and, hence, increasing the lifetime of a sensor network. We present an algorithm, which places copies of the requested data and updates them so as to minimize the communication overhead and power consumption of data transfer.

Our algorithm is completely distributed and requires very little local processing. The amount of bookkeeping involved is small, which fits in nicely with the constraint of limited memory resources. Also it makes minimal assumptions about the underlying MAC and routing layers, although pro-active routing algorithms like DSDV are not a good choice for these types of networks.

In conclusion, data placement is a new approach for energy conservation in wireless sensor networks. To our knowledge, very little previous work has been done to apply data placement to a location-aware network.

Further work such as accommodating quickly moving observers and accounting for node failures needs to be done to introduce guarantees into the data placement model. In the big picture, data placement may act as a service that aids in providing power saving and QoS guarantees to applications running on these sensor networks. This is analogous to how web-caching and content distribution help in providing better performance and guarantees over the Internet.

Acknowledgements

This work was supported in part by DARPA grant F33615-01-C-1905, MURI grant N00014-01-1-0576, and NSF grant CCR-0208769.

References

1. Lili Qiu, Venkata Padmanaban, Geoffrey M Voelker, On the Placement of Web Server Replicas, Proc. IEEE INFOCOMM 2001.
2. Chalermek Intanagonwiwat, Ramesh Govindan and Deborah Estrin, Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks, In Proceedings of the Sixth Annual International Conference on Mobile Computing and Networks (MobiCom 2000), August 2000, Boston, Massachusetts.
3. John Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. Building Efficient Wireless Sensor Networks with Low-Level Naming. In Proceedings of the Symposium on Operating Systems Principles (SOSP 2001), Lake Louise, Banff, Canada, ACM. October 2001.
4. Ya Xu, John Heidemann, and Deborah Estrin, Geography-informed Energy Conservation for Ad Hoc Routing, Proceedings of the Seventh Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2001), Rome, Italy, July 16-21, 2001.
5. N. Bulusu, J. Heidemann and D. Estrin, GPS-less Low Cost Outdoor Localization For Very Small Devices, IEEE Personal Communications, Special Issue on "Smart Spaces and Environments", Vol. 7, No. 5, pp. 28-34, October 2000.
6. Radhika Nagpal, Organizing a Global Coordinate System from Local Information on an Amorphous Computer, MIT AI Memo 1666, August 1999
7. Young-Bae Ko and Nitin H. Vaidya, "Location-Aided Routing(LAR) in Mobile Ad Hoc Networks," In Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 1998), ACM, Dallas, TX, October 1998.
8. C. E. Perkins and E. M. Royer, "Ad-hoc On Demand Distance Vector Routing." 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'99), New Orleans, Louisiana, February 1999.
9. Charles E. Perkins and Pravin Bhagwat, Highly dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers, in *SIGCOMM Symposium on Communications Architectures and Protocols*, (London, UK), pp. 212-225, Sept. 1994.
10. M. Charikar, S. Guha, E. Tardos and D.B. Shmoys, A constant factor approximation algorithm for the k-median problem. Proceedings of the 31st Annual ACM symposium on Theory of Computing.

11. M. Charikar and S.Guha, Improved Combinatorial Algorithms for the Facility Location and K-median Problems. In Proc. Of the 40th Annual IEEE Conference on Foundations of Computer Science, 1999.
12. N. Bulusu, J. Heidemann and D. Estrin, Adaptive Beacon Placement, Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21), Phoenix, Arizona, April 2001.
13. Tomasz Imielinski and Samir Goel, "DataSpace - querying and monitoring deeply networked collections in physical space," IEEE Personal Communications Magazine, Special Issue on Networking the Physical World, October 2000.
14. Philippe Bonnet, J. E. Gehrke, and Praveen Seshadri. "Querying the Physical World," IEEE Personal Communications, Vol. 7, No. 5, October 2000, pages 10-15. Special Issue on Smart Spaces and Environments.
15. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, " System architecture directions for network sensors," ASPLOS 2000.
16. Development platform for self-organizing wireless sensor networks, Proc. SPIE, Unattended Ground Sensor Technologies and Applications, Vol. 3713, p. 257-268.
17. A Compendium of NP optimization problems http://www.nada.kth.se/viggo/problrmlist/compendium.html
18. Self-organizing distributed sensor networks, Proc. SPIE, Unattended Ground Sensor Technologies and Applications Vol. 3713, p. 229-237.
19. The ns-2 simulator. http://www.isi.edu/nsnam.
20. Distributed Operating systems. Andrew S. Tanenbaum Prentice Hall.
21. Guillaume Pierre, Maarten van Steen, andAndrew S. Tanenbaum, *Self-replicating Web documents*, Technical Report IR-486, Vrije Universiteit, Amsterdam, February 2001,
22. The Case for Geographical Push Caching. *Proceedings of the Fifth Annual Workshop on Hot Operating Systems*, Orcas Island, WA, May 1995
23. E. M. Royer and C. E. Perkins, Multicast operation of the ad-hoc on-demand distance vector routing protocol, in Proc. of ACM/IEEE Intl. Conference on Mobile Computing and Networking (MOBICOM), Aug. 1999
24. A Survey of Multicast Technologies (2000), Vincent Roca, Luís Costa, Rolland Vida, Anca Dracinschi, Serge Fdida September 2000.
25. J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gumadi, S. Rhea, H. Weatherspoon, W. Weimer, C.Wells, B. Zhao. Oceanstore: An Architecture for Global-scale Persistent Storage. In the *Proceedings of ASPLOS 2000*, Cambridge, Mmassachusetts, Nov. 2000.
26. I. Stoica, R. Morris, D. Karger, f. Kaashoek, H. Balakrishnan. Chord: A Scalable Peer-to-Peer lookup Service for Internet Applications. In *Proceedings of ACM Sigcomm 2001*, San Diego, CA, Aug. 2001.
27. S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of ACM Sigcomm 2001,* San Diego, CA, Aug. 2001.
28. A. Woo, and D. Culler. A Transmission Control Scheme for Media Access in Sensor Networks, Proceedings of the Seventh Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2001), Rome, Italy, July 16-21, 2001.
29. B. Karp and H. Kung, Greedy Perimeter Stateless Routing, in Proc. of the Sixth Annual ACM/IEEE Intl. Conference on Mobile Computing and Networking (MOBICOM), Boston, 2000.
30. Yongqiang Huang, Hector Garcia-Molina. Publish/ Subscribe Tree Construction in Wireless Ad-Hoc Networks, 4th International Conference on Mobile Data Management, January, Melbourne, Australia, 2003.

Design and Implementation of a Framework for Efficient and Programmable Sensor Networks

Athanassios Boulis, Chih-Chieh Han, and Mani B. Srivastava
Networked and Embedded Systems Laboratory (NESL), EE Dept. UCLA
{boulis, simonhan, mbs}@ee.ucla.edu

Abstract **– Wireless ad hoc sensor networks have emerged as one of the key growth areas for wireless networking and computing technologies. So far these networks/systems have been designed with static and custom architectures for specific tasks, thus providing inflexible operation and interaction capabilities. Our vision is to create sensor networks that are open to multiple transient users with dynamic needs. Working towards this vision, we propose a framework to define and support lightweight and mobile control scripts that allow the computation, communication, and sensing resources at the sensor nodes to be efficiently harnessed in an application-specific fashion. The replication/migration of such scripts in several sensor nodes allows the dynamic deployment of distributed algorithms into the network. Our framework, SensorWare, defines, creates, dynamically deploys, and supports such scripts. Our implementation of SensorWare occupies less than 180Kbytes of code memory and thus easily fits into several sensor node platforms. Extensive delay measurements on our iPAQ-based prototype sensor node platform reveal the small overhead of SensorWare to the algorithms (less than 0.3msec in most high-level operations). In return the programmer of the sensor network receives compactness of code, abstraction services for all of the node's modules, and in-built multi-user support. SensorWare with its features apart from making dynamic programming possible it also makes it easy and efficient without restricting the expressiveness of the algorithms.***[*]

I. INTRODUCTION

Wireless ad-hoc sensor networks (WASNs) have drawn a lot of attention in recent years from a diverse set of research communities. Researchers have been mostly concerned with exploring applications such as target tracking and distributed estimation, investigating new routing and access control protocols, proposing new energy-saving algorithmic techniques for these systems, and developing hardware prototypes of sensor nodes.

Little concern has been given on how to actually program the WASN. Most of the time, it is assumed that the proposed algorithms are hard-coded into the memory of each node. In some platforms the application developer can use a node-level OS (e.g. TinyOS) to create the application, which has the advantages of modularity, multi-tasking, and a hardware abstraction layer. Nevertheless the developer still has to create a single executable image to be downloaded manually into each node. However, it is widely accepted that WASNs will have long-deployment cycles and serve multiple transient users with dynamic needs. These two features clearly point in the direction of dynamic WASN programming.

What kind of dynamic programmability do we want for WASNs? Having a few algorithms hard-coded into each node but tunable through the transmission of parameters, is not flexible enough for the wide variety of possible WASN applications. Having the ability to download executable images into the nodes is not feasible because most of the nodes will be physically unreachable or reachable at a very high cost. Having the ability to use the network in order to transfer the executable images to each and every node is energy inefficient (because of the high communication costs and limited node energy) and cannot allow the sharing of the WASN by multiple users. What we ideally want is to be able to dynamically program the WASN as *a whole, an aggregate*, not just as a mere collection of individual nodes. This means that a user, connected to the network at any point, will be able to inject instructions into the network to perform a given (possibly distributed) task. The instructions will task individual nodes according to user needs, network state, and physical phenomena, *without any intervention from the user*, other than the initial injection. Furthermore, since we want multiple users to use the WASN concurrently, several resources/services of the sensor node should be abstracted and made sharable by many users/applications.

One approach of programming the WASN as an aggregate is a distributed database system (e.g., [21]). Multiple users can inject database-like queries to be autonomously distributed into the network. The WASN is viewed as a distributed database and the query's task is to retrieve the needed information by finding the right nodes and possibly aggregate the data as they are routed back to the user. This approach ignores though the fact that information is not always resident in nodes but sometimes has to be retrieved by *custom collaboration* among a changing set of nodes (e.g., target tracking). Thus even

[*] This work was partially supported by DARPA SensIT program, by ONR MinuteMan project, and by the NSF funded UCLA Center for Embedded Networked Sensing.

though the database model is programming the network in the desirable way, it is not expressive enough to implement any distributed algorithm.

The other approach to WASN programmability that is used by our framework, and is gaining momentum lately, is the "active sensor" approach. This term was used in [20], to describe a family of frameworks that try to task sensor nodes in a custom fashion, much like active networking frameworks task data network nodes. The difference is that while active networking tasks are reacting only to reception of data packets, active sensor tasks need to react to many types of events, such as network events, sensing events, and timeouts. Active sensor frameworks abstract the run-time environment of the sensor node by installing a virtual machine or a high-level script interpreter at each node. For example, single instructions of the scripts (or bytecodes) can send packets, or read data from the sensing device. Moreover, the scripts (or bytecodes) are made mobile through special instructions, so nodes can autonomously task their peers.

The difficulty in designing an active sensor framework is how to properly define the abstraction of the run-time environment so that one achieves compactness of code, sharability of resources for multi-user support, portability in many platforms, while at the same time keeping a low overhead in delays and energy. Our proposal of such a framework, called SensorWare, employs lightweight and mobile control scripts that are autonomously populated in sensor nodes after a triggering user injection. The sensor node abstraction was made in such a way so that multi-user accessibility is given to all of the node's modules (e.g., radio, sensing devices) while also creating other services (e.g., real-time timers). Considerable attention was given to the portability and expandability of the framework by allowing the definition of new modules. By choosing the right level of abstraction the scripts are compacted to 10s-100s of bytes. For the non-trivial application examined in section V.A, the SensorWare script is smaller than the code of other frameworks with comparable capabilities in algorithm expressiveness (e.g. other active sensors scripts, binary images).

Our implementation and porting of SensorWare in several sensor node platforms shows that the size of the framework is small enough (<180Kbytes) to fit in most current sensor node designs. Moreover, extensive measurements in our prototype iPAQ-based sensor node platform reveal the delay and energy overheads of SensorWare. Every SensorWare script command has a delay less than 0.3msec showing the limits of real-time operation. Note that the script commands have a high-level of abstraction (i.e., each command performs multiple low-level operations). Experiments with both compiled and interpreted versions of the scripts are conducted in order to explore the energy trade-off space between different representations of an algorithm.

Section II discusses in depth the nature of WASNs, approaches to WASN programmability, and the general idea of our approach. Section III presents related work. Section IV presents SensorWare's architecture. Section V illustrates how is SensorWare ported to a platform and explains a moderately large script solving a real problem. Section VI presents our current implementation and the measurements we acquired through it. Finally, section VII concludes the paper.

II. Motivation and Background

A. *Wireless Ad hoc Sensor Networks*

Figure 1 shows an example of a WASN, highlighting its main characteristics. An ad hoc network of miniature, resource-limited, static, wireless, sensor nodes is being used to monitor a dynamic physical environment. The use of low power communication and the need for diversity in sensing necessitates a multi-hop, distributed architecture [24]. Typically a user queries the network (consider the term "query" in the broad sense, not just database query), the query triggers some reaction from the network, and as the result of this reaction the user receives the information needed. The reaction to the query can vary from a simple return of a sensor value, to a complex unfolding of a distributed algorithm among some or all of the sensor nodes, such as a collaborative signal processing algorithm or a distributed estimation algorithm. Furthermore, there are multiple users who are transiently connected to the network; each having different needs in requested information.

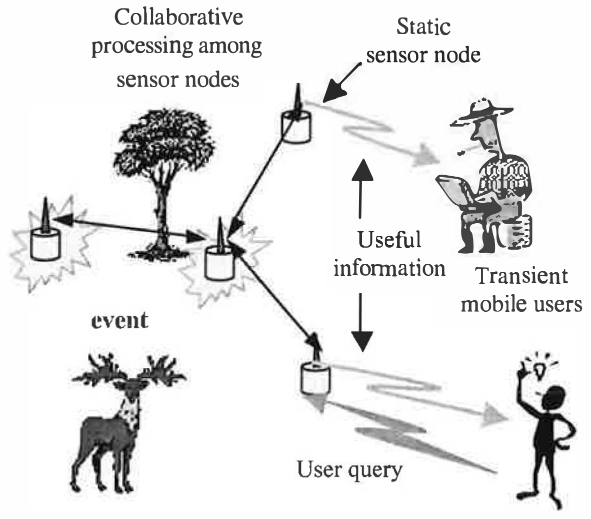


Figure 1: Wireless Ad-hoc Sensor Network

These systems are quite different from traditional networks. First, they have severe energy, computation, storage, and bandwidth constraints. Second, their overall usage scenario is quite different from traditional networks. There is not a mere exchange of data between users and nodes. The user will rarely be interested in the readings of one or two specific nodes. The user will be interested in some parameters of a dynamic physical process. To efficiently achieve this, the nodes have to form an application-specific distributed system to provide the user with the answer. Moreover, the nodes that are involved in the process of providing the user with information are constantly changing as the physical phenomenon is changing. Therefore the user interacts with the system as a whole. The WASN is not there to connect different parties together as in the traditional networking sense but to provide information services to users.

As a consequence, efficiently designed WASNs operate in a fashion where a node's actions are affected largely by physical stimuli detected by the node itself or nearby nodes. Frequent long trips to the user are undesirable because they are time and energy consuming. This decentralized (i.e. not all traffic flows to/from user), autonomous (i.e., user out-of-the-loop most of the time) way of operating, is called "proactive computing" (as opposed to interactive) by David Tennenhouse [29]. We also adopt the term "proactive" throughout the paper to denote an autonomo us and non-interactive nature.

Efficiently designed WASNs are application-specific distributed systems that require a different distributed proactive algorithm as an efficient solution to each different application problem. Given the nature of SNs, one can coarsely define two classes of problems in their design. First, the application-specific problem: How does one find the most efficient distributed algorithm for a particular problem? Second, the generic problem: How does one dynamically deploy different algorithms into the network, what is the programming model that will implement these algorithms, and what general support does one need from the framework? The second class of problems is the focus in this paper, emphasizing in the description of our own framework, i.e., SensorWare.

B. Approaches to WASN programmability

As mentioned in the introduction, a popular approach to dynamic WASN programmability views the WASN as a distributed database. The data exist in the network and have to be found, probably processed in predefined ways (e.g., aggregated) and delivered to the user. Heidemann et al. [10], closely follow this model without explicitly employing traditional database terms and mechanisms. They focus on a data-driven low-level naming scheme based on attributes. A query describes the data it is looking for and directed diffusion [15] is used as the underlying routing protocol. The data can be processed with predefined filters as they are routed back to the user. Other systems, such as Cougar [1], focus more on transferring the SQL semantics of traditional databases to the distributed setting of WASNs. In this case, the naming system developed in [10] is replaced by an SQL equivalent. Each node is equipped with a fixed database query resolver. As queries arrive to a node, the local resolver decides on the best, distributed plan to execute the query and distributes the query to the appropriate nodes. The more recent and probably more advanced system that follows the database model is TinyDB [21] developed in Berkeley. Their main focus is aggregate queries (e.g., min, max, avg) thus they provide special optimizations for them (e.g., exploit the shared medium, perfonn hypothesis testing).

The strong point of the database approach is that it offers an intuitive way to extract information from a WASN hiding the complications of *embedded and distributed* programming. The model's limitation though is that there are only pre-defined ways to process the data, which implies that only certain types of applications (i.e. applications that were studied by the specific researchers and are mainly aggregation applications) are addressed in the most efficient way by the database model. If a new way to process and react to the data is needed by application N&U (New-and-Unexplored), this can only be done at the user node (assuming that the human-controlled user node is easily upgradeable). Consequently, the algorithmic pattern to address application N&U under the database model will be an iteration of the generalized steps: 1) partially processed data arriving to the user node, 2) data undergoing custom processing and 3) based on the result a new database query is issued. In most cases, this is not the structure of the most efficient algorithm to solve an application problem.

C. SensorWare

Our proposal seeks to remedy the limited fexibility problem at the expense of increased responsibility for the programmer. SensorWare provides a language model powerful enough to implement any distributed algorithm while at the same time hiding unnecessary low-level details from the application programmer and providing a way to share the resources of a node among many applications and users that might concurrently use the WASN. A distributed algorithm can be viewed as a set of collaborating programs executing in a corresponding (often time-varying) set of nodes. In SensorWare these programs are sensor-node control scripts. The sensing, communication, and signal-processing resources of a node are exposed to the control scripts that orchestrate the dataflow to assemble custom protocol and signal processing stacks.

Equally important is the role of SensorWare in the dynamic deployment of the distributed algorithms into the network. Usually this means that a distributed algorithm has to be incorporated in several sensor nodes, which in turn means that these sensor nodes have to be dynamically programmed. A user-friendly and energy-efficient way of programming the nodes keeps the user out-of-the-loop most of the time by allowing sensor nodes to program their peers. By doing so, the user does not have to worry about the specifics of the distributed algorithm (because the information on how the algorithm unfolds lies within the algorithm), and the nodes save communication energy (because they interact with their immediate neighbors and not with the user node through multi-hop routes). In order to facilitate the user-friendly and energy-efficient dynamic deployment of an algorithm the scripts are made mobile using special language commands and directives. A script can replicate or migrate its code and data to other nodes, directly affecting their behavior. The replication or migration of a script will be called "population" in the paper. The user "injects" the query/program into the network, and the query *autonomously* unfolds the distributed algorithm into the nodes that should be affected.

A usage scenario can be like the following: A user sends a query to the sensor network. The query is a script, a state machine in its simplest form, which is injected to one or more sensor nodes. The script will describe among other things how it is going to populate itself to other nodes. The process of population can continue depending on events and the current state. For example as the events of interest are moving to a different area, the scripts can move along with them, possibly trying to predict their next move. The populated scripts will collaborate among themselves in order to extract the information needed by the user, and when this information is acquired it is sent back to the user. Although the scripts are defining behavior at the node level, SensorWare is not a node-level programming language. It can be better viewed as an event-based language since the behaviors are not tied to specific nodes but rather to possible events that depend on the physical phenomena and the WASN state.

It should be also noted that this model comes at a cost, compared to the database model. The programmer has to explicitly take care of the distribution of the algorithm, so only the complications of *embedded* programming are hidden.

III. RELATED WORK

SensorWare falls under the family of active sensor frameworks. Its closest relatives in the traditional networks realm are Mobile Agent frameworks. Other active networking frameworks exhibit similarities, such as the scripting abstraction. In this section we only consider work that tries to make WASNs programmable using active sensor concepts. Therefore, general mobile-agent and active-network platforms are not presented, nor any distributed database systems for WASNs are further discussed. The interested reader can refer to [2] for a comprehensive comparison of SensorWare with mobile agent platforms, as well as with an active networking framework called PLAN [11].

An active sensor framework for WASNs is currently being developed in Berkeley under the name Maté. Maté [20] is a tiny virtual machine build on top of TinyOS [13]. TinyOS is an operating system, designed specifically for the Berkeley-designed family of sensor nodes, generically named "motes" [12][13]. Maté's goal is to make a WASN made of motes dynamically programmable in an efficient manner. This includes the capability to dynamically instruct a mote to execute any program, and expressing this program in a concise way. They achieve this by building a virtual machine (VM) for the motes. The virtual machine supports a very simple, assembly-like language, to be used for all needs of mote-tasking. Programs (called capsules) written on the VM language can be injected to any node and perform a task. Furthermore the capsules have the ability to self-transfer themselves by using special language commands. This model seems extremely like our own in SensorWare. Indeed, Maté shares the same goals as SensorWare as well as the same basic principles to achieve these goals. Differences appear though when one looks thoroughly into each platform's implementation.

Maté, like its substrate TinyOS, was built with a specific platform in mind: the extremely resource-limited mote. The main restriction for the developer of mote-targeted frameworks (such as an OS or a VM) is memory. The newest version of a mote called mica offers 128Kbytes of program memory and 4Kbytes of RAM. An older version called rene2 has 16Kbytes of program memory and 1Kbyte of RAM. Maté, with an ingenious architecture, supports both platforms. Being so memory constrained, Maté has to sacrifice some features that would make programming easier and more efficient. First, a stack-based architecture with an ultra-compact instruction set (all instructions are 1 byte) is adopted which is reminiscent of a low-level assembly language or the byte code of the Java VM. This kind of model makes programming of even medium-sized tasks difficult. Furthermore, due to the ultra-compact instruction set, many 1-byte instructions are needed to express a medium complexity algorithm, which in turn leads to large programs, compared to a higher-level, more abstracted scripting language. The size of programs is important since the code is transmitted/received using the radios of the nodes spending energy for every transmitted/received bit. Second, the behavior of a program when radio packets are received is rather rigid. A handler to process such events is essentially stateless in Maté. Thus, if a new pattern of packet processing is needed, a new handler has to be transferred through the network. This imposes an overhead in energy consumption and execution time. Third, because there is only one context (i.e., handler) per event (e.g., clock tick, reception of packet) multiple applications cannot run concurrently in one mote.

SensorWare cannot fit in the restricted memory of a mote. SensorWare targets richer platforms that we believe are going to be the mainstream in sensor node design in the immediate future. Such platforms (e.g., [26]) include a 1Mbyte of program memory and 128Kbytes of RAM. Having the luxury of more memory, SensorWare supports easy programming with a high-level scripting language, as well as concurrent multi-tasking of a node so that multiple applications can concurrently execute in a WASN. The programming model and properties of SensorWare are extensively discussed in section IV.

Particularly instructive is to study the relationship between SensorWare's mobile scripting approach and the mobile code approach in Penn State's Reactive Sensor Network [25] (RSN) project under DARPA's SenseIT program [27]. RSN's focus is on providing an architecture whereby sensor nodes can: (i) download executables and DLLs, identified by URLs, from repositories or their cache, (ii) execute the program at the local node using input data which itself may be remotely located and identified by a URL, and (iii) write the data to a possibly remote URL. The RSN model is in essence Java's applet model generalized to arbitrary executables and data, and combined with a lookup service. The focus of RSN is quite different from SensorWare. Differences include: (i) RSN provides a general lookup and download service, (ii) RSN does not seek to provide a scripting environment with an associated sensor node resource model for use by scripts, and (iii) RSN's notion of mobility is download oriented, as opposed to SensorWare's approach of a script which can autonomously spawn scripts to remote nodes. RSN views sensor nodes as network switches with dynamically adaptable protocols, trying to directly map the motivation and methods of classical active networks into sensor networks. Unfortunately such an approach does not address the basic problems of sensor networks. Although one might be able to construct some distributed applications using the above scheme, by no means the creation and diffusion of distributed proactive applications into the network is supported by its architecture.

Finally, extremely relevant is the work that is being conducted in University of Delaware by Jaikaeo et al. [17] called SQTL (Sensor Querying and Tasking Language). Having the same goals as our research, but starting from a different point (database-like queries), the researchers end up with the same basic solution as SensorWare, namely a tasking language for sensor networks. To lively demonstrate the relevance to our work we are quoting an excerpt from [17]. *"We model a sensor network as a set of collaborating nodes that carry out querying and tasking programmed in SQTL. A frontend node injects a message, that encapsulates an SQTL program, into a sensor node and starts a diffusion computation. A sensor node may diffuse the encapsulated SQTL program to other nodes as dictated by its logic and collaboratively perform the specified querying or tasking activity."*

SQTL fits in a more general architecture for sensor networks called SINA (Sensor Information Networking Architecture) [28]. SINA uses both SQL-like queries as well as SQTL programs. Some of its main features include: 1) hierarchical clustering, 2) attribute-based naming, 3) a spreadsheet paradigm for organizing sensor data in the nodes. SQL-like queries use these three features to execute simple querying and monitoring tasks. When a more advanced operation is needed though, SQTL plays the essential role by programming (or "tasking" as the researchers from Delaware call it) the sensor nodes and allowing proactive population of the program. In SINA, SQTL is used as an enhancement of simple SQL-like queries. The framework is there mainly to support the queries not the mobile scripts. As a consequence, SQTL scripts do not have all the provisions that SensorWare scripts have. The most important of them are: 1) Rich sensor-node-related APIs (e.g. for networking, sensing). 2) Diverse rules for mobility. A SQTL script can only specify the nodes to be populated. SensorWare first checks if the script is already in the remote node and offers a multitude of possibilities depending on how many instances of the script are already running in the remote node. 3) Code modularity in order to share functionality and avoid redundant code transfers 4) Support for multi-user scripts. 5) Resource management in the presence of multiple scripts running in the node.

IV. Architecture

First, we show SensorWare's place inside the overall sensor node's architecture (Figure 2). The architecture of a sensor node can be viewed in layers. The lower layers are the raw hardware and the hardware abstraction layer (i.e., the device drivers). An operating system (OS) is on top of the lower layers. The OS provides all the standard functions and services of a multi-threaded environment that are needed by the layers above it. The SensorWare layer for instance, uses those functions and services offered by the OS to provide the run-time environment for the control scripts. The control scripts rely completely on the SensorWare layer while populating around the network. Static applications and services coexist with mobile scripts. They can use some of the functionality of SensorWare as well as standard functions and services of the OS. These applications can be solutions to generic sensor node problems (e.g., location discovery), and can be distributed but not mobile. They will be part of the node's firmware.

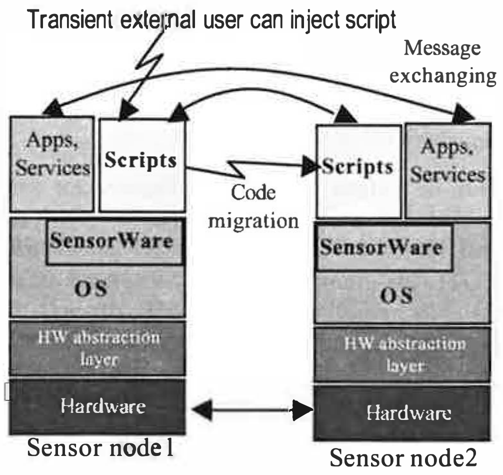


Figure 2: The general sensor node architecture

Two things comprise SensorWare: 1) the language, and 2) the supporting run-time environment. The next two subsections describe each of the parts in detail. A third subsection discusses issues of portability and expandability, and presents the final SensorWare code structure. Finally, the fourth subsection discusses the issues of addressing and routing in SensorWare.

A. The language

As discussed earlier, the basic idea is to make the nodes programmable through mobile control scripts. Here the basic parts that comprise the language will be described as well as the programming model that emerges from the parts.

First, a scripting language needs proper functions/commands to be defined and implemented in order to use them as building blocks (i.e., these will be the basic commands of the scripts). Each of these commands will abstract a specific task of the sensor node, such as communication with other nodes, or acquisition of sensing data. These commands can also introduce needed functionality like moving a script to another node or filtering the sensing data through a filter implemented in native code. Second, a scripting language needs constructs in order to tie these building blocks together in control scripts. Some examples include: constructs for flow control, like loops and conditional statements, constructs for variable handling and constructs for expression evaluation. We call all these constructs the "glue core" of the language, as they combine several of the basic building blocks to make actual control scripts.

Figure 3 illustrates the different parts of the SensorWare language. Several of the basic commands/functions are grouped in theme-related APIs. We use the term API in a generic fashion, to denote a collection of theme-related functions that provide a programming interface to a resource or a service. As the figure hints, there is a question on what happens when we are dealing with different sensor node platforms that may support different/additional kinds of modules. Do we allow the set of APIs to be expandable? If so, who has the authority to name and define new commands? We will return to this topic with a solution in subsection C.

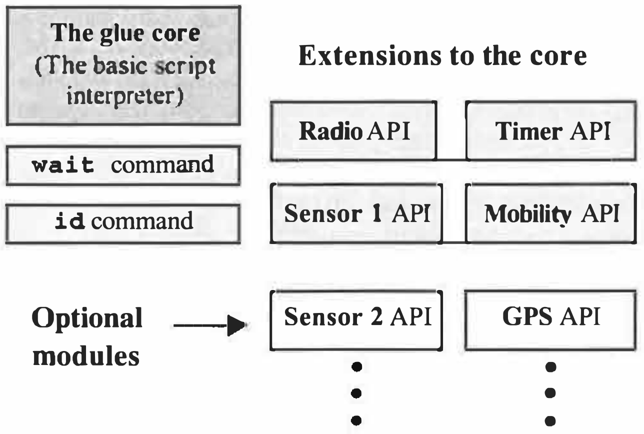


Figure 3: The language parts in SensorWare

As a glue core we can use the core from one of the scripting languages that are freely available, so we are not burdened with the task of building and verifying a core. One such scripting language, that is well suited for SensorWare's purposes, is Tcl [22], offering great modularity and portability. Thus, the Tcl core is used as the glue core in the SensorWare language. All the basic commands, such as `wait`, or the ones included in the APIs, are defined as new Tcl commands using the standard method that Tcl provides for that purpose.

The set of APIs is basically a way of easily exporting services and shared resources to the scripts. For example, the Timer API defines and sets/resets real time timers, while the Mobility API provides the basic functions to the scripts so they can transfer themselves around the network.

A.1 The general programing model

As discussed earlier, according to the proactive distributed model the scripts will look mostly like state machines that are influenced by external events. Such events include network messages from peers, sensing data, and expiration of timers. The programming model that is adopted is equivalent to the following: An event is described, and it is tied with the definition of an event handler. The event handler, according to the current state, will do some (light) processing and possibly create some new events or/and alter the current state. Figure 4 illustrates SensorWare's programming model with an example.

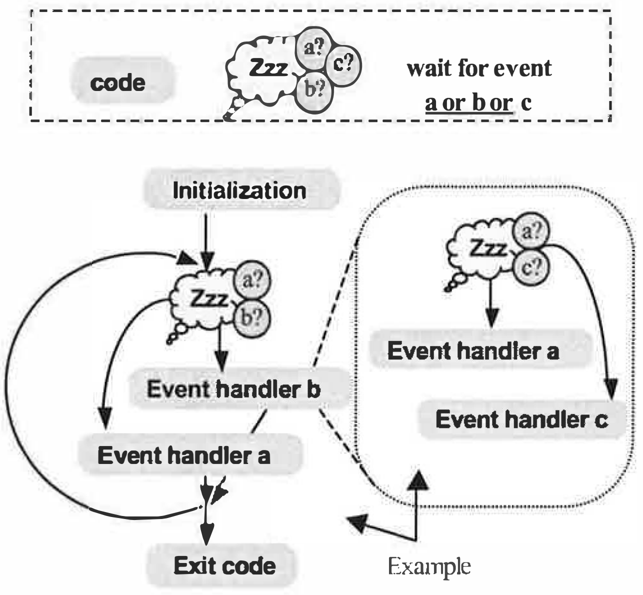


Figure 4: The programming model

The behavior described above is achieved through the `wait` command. Using this command, the programmer can define all the events that the script is waiting upon, at a given time. Examples of events that a script can wait upon are: i) reception of a message of a given format, ii) traversal of a threshold for a given sensing device reading, iii) filling of a buffer with sensing data of a given sampling rate, iv) expiration of several timers. When *one* of the events declared in the `wait` command occurs, the command terminates, returning the event that caused the termination. The code after the `wait` command processes the return value and invokes the code that implements the proper event handler. After the execution of the event handler, the script moves to a new `wait` command, or more usually it loops around and waits for events from the same `wait` command.

B. The run-time environment

Equally important to the programming model is the run-time environment that supports the scripts. Figure 5 illustrates the basic tasks performed by the environment. We separate tasks into fixed and platform-specific. The fixed tasks are always included in a SensorWare implementation, while the platform-specific depend on the existence of specific modules and services in the node platform. Again, the problem of expandability and portability appears. Do we allow any developer to *arbitrarily* define and create any tasks, according to the specific needs of each platform? Subsection C addresses this question. The Script Manager is the task that accepts all requests for the spawning of new scripts. It forwards the request to the Admission Control task and upon receiving a positive reply, it *initiates a new thread/task running a script interpreter for the new script*. The Script Manager also keeps any script-related state such as script-data for as long as the script is active.

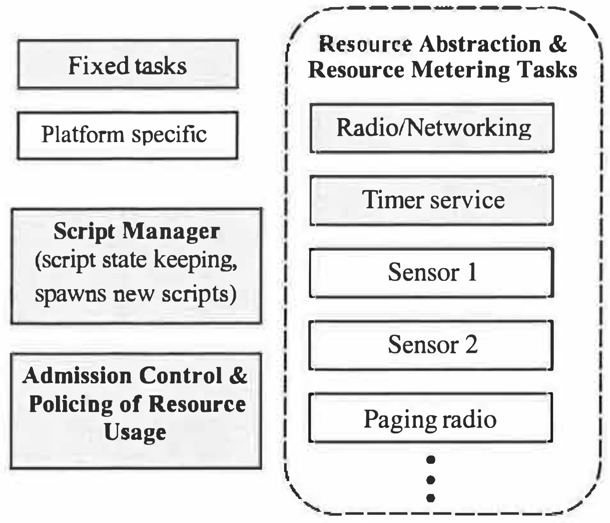

Fixed tasks
Platform specific
Script Manager (script state keeping, spawns new scripts)
Admission Control & Policing of Resource Usage
Resource Abstraction & Resource Metering Tasks
Radio/Networking
Timer service
Sensor 1
Sensor 2
Paging radio

Figure 5: Tasks in the SensorWare run-time environment

The Admission Control and Policing of Resource Usage task, as the name reveals, takes all the script admission decisions, makes sure that the scripts stay under their resource contract, and most importantly checks the overall energy consumption. If the overall consumption exhibits alarming characteristics (e.g., the current rate cannot support all scripts to completion) the task selectively terminates some scripts according to certain SensorWare policies. For more information on resource management the interested reader can refer to [4].

The run-time environment also includes "Resource Abstraction and Resource Metering" tasks (sometimes referred to as "Resources Handling" tasks for brevity). Each task supports the commands of the corresponding APIs and manages a specific resource. There are two fixed tasks in this category since every platform is assumed to have at least one radio and a timer service. The "Radio" task manages the radio: i) it accepts requests from the scripts about the format of network messages that they expect, i) it accepts all network messages and dispenses them to the appropriate scripts according to their needs, and finally iii) measures the radio utilization for each script, a quantity that is needed by the "Admission Control & Policing of Resource Usage" task. The second fixed task, the "Timer service", accepts the various requests for timers by all the scripts and manages to service them using a real-time timer the embedded system provides. In essence the task provides many virtual timers relying on one timer provided by the system. According to platform capabilities a specific porting of SensorWare may run additional tasks For instance, a "Sensor Abstraction" task manages a sensing device. It accepts all requests for sensor data from all the scripts and decides on the optimal way to control the sensing device (e.g., setting the A/D sampling rate). It also measures the sensing device utilization for each script. Figure 6 depicts an abstracted view of SensorWare's run-time environment for an example platform with one sensing device.

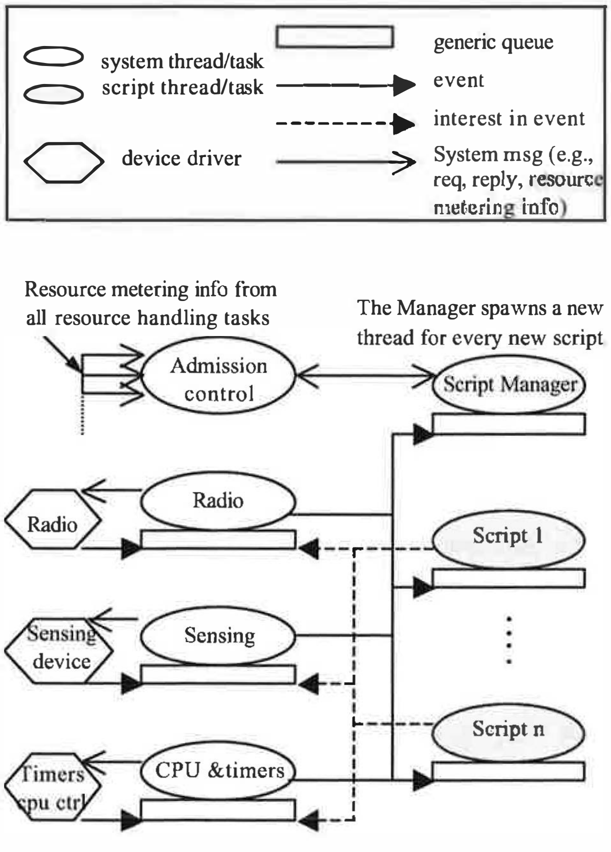


Figure 6: Abstracted view of SensorWare's run-time environment for an example platform

Most of the threads running are coupled with a generic queue. Each thread "pends" on its corresponding queue, until it receives a message in the queue. When a message arrives it is promptly processed. Then the next message will be fetched, or if the queue is empty, the thread "pends" again on the queue. A queue associated with a script thread is receiving events (e.g., reception of network messages, sensing data, or expiration of timers). A queue associated with one of the resource handling tasks, receives events of one type (from the specific device driver that is connected to), as well as messages that declare interest in this event type. For instance, the Sensing resource-handling task is receiving sensing data from the device driver and interests on sensing data from the scripts. The Script Manager queue receives messages from the network that wish to spawn a new script. There are also system messages that are exchanged between the system threads (like the ones that provide the Admission Control thread with resource metering information, or the ones that control the device drivers).

Finally, concerning security, we distinguish between code safety and security in the following sense: code safety relates to the execution of a script in the SensorWare run-time environment inside a node, whereas security relates to the network as a whole. For code safety, one would want guaranties that a buggy or malicious script will not have any effect on other scripts or on the run-time system. For security, one would want guaranties that an intruder could not gain access to resources or information of the network, and could not affect the use of the network by legitimate users. SensorWare does not consider general security issues. The major problems are authenticating the current set of users and deny any service to anyone else, as well as encrypt the data. Wen et al. [31] describe a security scheme for sensor networks called SPINS that could work alongside with SensorWare. Code safety on the other hand is an integral part of SensorWare as it is closely related to the language and run-time environment design choices. For more information on SensorWare's code safety the reader can refer to [2].

C. *Portability and expandability of SensorWare*

In the previous subsections the problem of platform variability was revealed. Here we will present a solution for SensorWare's code structure. There are two kinds of platform variability: 1) capabilities variability (i.e. having different modules, such as sensing devices, GPS), 2) HW/SW variability (i.e. although the capabilities are the same we have different OS and/or specifics of hardware devices). We will explore solutions for each kind in two different subsections.

C.1 Capabilities variability

Different platforms may have different capabilities. For instance, imagine that one platform A has a radio and a magnetometer, while another platform B has two radios (a normal and a paging one) and a camera. How will we abstract the two platforms with the same framework? Since SensorWare's building blocks are the interface to the abstracted modules/services, we can allow an expandable API. Further, most modules/services will need a supporting task (as described in subsection B), so we can allow the definition and addition of arbitrary tasks in SensorWare's run-time environment. This kind of solution would create severe problems in the manageability of the code by different developers. SensorWare advocates a more modular and well-structured solution. SensorWare declares, defines, and support virtual devices (an idea triggered by Linux's virtual devices). Any module or service is represented as a virtual device. For example a radio, a sensing device, the timer service, a location discovery protocol are all view as virtual devices.

There is a **fixed interface** for all devices. More specifically there are four commands that are used to communicate with the device. They are: `query`, `act`, `createEventID`, and `disposeEventID`. `Query` asks for a piece of information from the device and expects an immediate reply. `Act` instructs the device to perform an action (e.g., modify some parameters of the device, or if the device is an actuator perform an action). `CreateEventID` describes a specific event that this device can produce and gives this event a name/ID. The name can be used subsequently from the `wait` command to wait on this specific event. `DisposeEventID` just disposes that name. Additionally, if a device can produce events, a task is needed to accept createEventID and disposeEventID commands and react to wait commands that are waiting on the device's events. The task definition, and the parsing of the arguments of the four commands are defined in a custom fashion by the developer. This is where the expandability stems from, while at the same time keeping a structured form.

C.2 HW/SW variability

Even though two platforms may have the same capabilities (i.e., the same modules/services), they may rely on different hardware and/or operating system. In order to facilitate the porting process it is desirable to clearly separate the OS and HW-specific code from the fixed code and the capabilities-definition code. To achieve this we need to identify the dependencies of the code to the OS and the hardware and create abstracted wrapper functions. The wrapper functions are actually defined in separate sections of the code (i.e., different .c files) so that the developer can easily identify the points of change for a porting procedure.

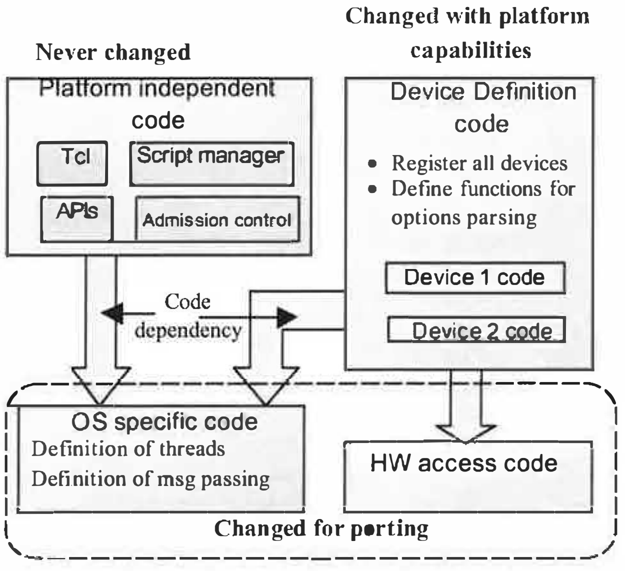


Figure 7:SensorWare code structure

From the OS we need support to create and initiate threads/tasks, and support to define, post, and pend into mailboxes/queues. Thus, we create wrapper functions for these operations. We also need low-level functions to access the hardware, thus we create wrapper functions around them (these functions will depend on the specific capabilities the platform supports). Figure 7 illustrates SensorWare's code structure.

D. Addressing and routing

Addresses in SensorWare have the generic format: [nodes_specification.script_name.userID.appID]. Currently nodes_specification is just a node ID but we are extending it to attribute-described nodes. Script_name is a string with a hierarchical structure: [$\text{name}_{\text{level\_0}}$. $\text{name}_{\text{level\_1}}$.... $\text{name}_{\text{level\_n}}$]. UserID is just a user id. AppID is an id denoting the application (i.e. collection of scripts) that the particular instance of a script belongs to. It is also used to distinguish instances of the same script running under the same node and under the same user (but for different applications).

Although a fixed addressing scheme is necessary we cannot say the same for a fixed routing scheme. Routing can take multiple facets in a WASN (e.g. directed diffusion, geographical routing, energy aware unicast, multicast to members of a cluster, etc). All these examples can be used by different applications or even by the same application according to circumstances. Furthermore, many applications can use their own custom-made routing, or more frequently, no routing at all, as they are restricted to purely 1-hop local interaction (e.g., the aggregation application we describe in the paper). Thus, SensorWare needs to provide a way to easily export the functionality of multiple routing protocols to the scripts and allow the easy insertion of new routing protocols at SensorWare compile time. The clearest way to achieve this is to define routing protocols as devices in SensorWare. Furthermore, in order to support application-level routing we define a special device that gives scripts the ability to handle system-kept routing tables, so they are alleviated from this burden.

V. CODE EXAMPLES

In order to make SensorWare more concrete, we will present code examples and porting details in the next two subsections. The first one involves the creation of a specific application using the SensorWare script language. The second example, present details on how to port SensorWare in a specific platform. More specifically, we will show how to define new devices and how to connect the framework with the existing OS and hardware.

A. Script example

In this subsection we will present the code for the snapshot aggregation application with multiple (static) users support. The specific problem that we are solving is to find the global maximum among current sensor node readings and report it back to the user. Furthermore, multiple users may request this maximum while the algorithm is running (i.e., time to populate the script into the network, collect and aggregate data towards the user). The users are accommodated with the minimum traffic, without the need to launch a different application/script for each user. Finding the minimum, average, or any other aggregation function, among different kinds of sensor node readings or state, can be easily achieved by trivial modification in our script. More on aggregation applications in general can be found in [3].

Before proceeding with the script code, it is beneficial to describe the internal workings of two Sensorware commands, namely "`replicate`" and "`wait`". `Replicate` (possibly) transfers the script that it was called from, to other node(s). It does not blindly pack and transmit the code and state of the script like analogous commands of other active sensor approaches currently do. `Replicate` first starts with a transmission of "intention to replicate" message, carrying the name of the script and the issuing user. If the same script already exits in the other node(s) `replicate`, according to options, defined by the user, may choose not to transfer the code, may choose to initiate a second script of the same type in the node, or if the script has multi-user support, send an "add user" message. By default, `replicate` will send the "intention to replicate" message to avoid unnecessary code transfers, and will spawn a second script only if the requesting user is different by the existing one. Furthermore, it is assumed by default that the parent of the script (i.e., the node that spawned the script to the current node) already has the code for the script, thus does not need an "intention to replicate" message. The arguments of the `replicate` command are:

replicate [-[f] [d] [p] [m] [rc] [rs] [ru]] [node_list]
[ ] means optional
f : forced replicate, no "intention to replicate" message sent
d: duplication of script at remote node irrespective of user
p: parent not assumed to have script in memory
m: script supports multi-users. Do not spawn new script in
remote node, instead send "add user" message to existing script
rc: return nodes that code was transferred
rs: return nodes that spawned new script
ru: return nodes "add user" message was sent
by default option rsru is in effect.
node_list: nodes to replicate. Leaving this field empty implies a broadcast to neighbors. Parent is excluded unless p is chosen.

It is also useful to reveal some of the details of the `wait` command. `Wait` returns when an event named in the command's arguments occurs. In order to expedite processing of the event by the subsequent scrip code, the wait command sets the following predefined variables:

event_name: the name of the occurred event. It indicates the device that caused the event and the type of the event
event_data data returned by the event
If the event is a packet reception the following are defined and set: ***msg_sender***, ***msg_body***

Listing 1 shows the actual SensorWare script. SensorWare commands and reserved words are in boldface. Variable names are in italics. Reserved variable name are in boldface and italics. Basic Tcl knowledge is needed to follow the script, although we do explain most of the code step by step. The example is sufficient to illustrate the programming style and the use of some of the most important commands, while solving a real problem.

```
set need_reply_from [ replicate -m]
set maxvalue [ query sensor value]
if {$need_reply_from == ""} { send $parent $maxvalue; exit}
else{ set return_reply_to $parent}
set first_time 1
while{1} {
    waitanyRadioPck // "anyRadioPck" is a predefined eventID
    If{ $msg_body==add_user} {
        if{ $first_time == 1 } {
                send $parent $msg_body
                set first_time 0
        }
        set return_reply_to "$return_reply_to $msg_sender"
    }else{
        set maxvalue [expr {($maxvalue<$msg_body) ? $maxvalue
: $ msg_body}]
        set n [lsearch $need_reply_from $ msg_sender]
        set need_reply_from [lreplace $need_reply_from $n $n]
    }
    foreach node $return_reply_to{
        If{ ($need_reply_from=="")||($need_reply_from==$node)} {
                send $node $maxvalue
                set n [lsearch $return_reply_to $node]
              set return_reply_to [lreplace $return_reply_to $n $n]
        }
    }
    if {$return_reply_to==""} {exit}
}
```

Listing 1: Multi-user aggregation code

The specific script keeps two important variables at each node: a list of nodes that replies are needed from, and a list of nodes that replies are due. The first command tries to replicate the script to all the neighbors (except the parent), declaring that this is a multi-user script. The nodes that the script was spawned or an "add user" message was sent are returned and added to the need_reply_from variable. The second command reads the current value from the sensing device and sets the maxvalue variable with it. If there are no nodes to return a reply the script sends the maxvalue to the parent node and exits. Otherwise the parent node is added to the list return_reply_to and the big loop begins. Each time a packet is received we check if it is a data reply or an "add user" message and modify our lists and maxvalue accordingly. To graphically see how this algorithm works, refer to [3].

The script is its raw form is 882 bytes. If reserved words and variable names are compressed, the script becomes 277 bytes. If furthermore, we compress this intermediate form with gzip, we end up with 209 bytes. This is a compact description for this non-trivial algorithm. An equivalent SQTL script has a size in the order of 1000 bytes (based on the simpler algorithm of aggregation for a single user and without replication checking). Building the same algorithm in Maté was proven impossible due to its limited heap and stack sizes. There was not enough space to hold the need_reply_from and return_reply_to lists. Even with a larger memory space though, Maté's stack based architecture and lack of higher-level services results in code of many instructions even for simple tasks. As stated earlier, Maté's restrictions are a design choice, coming from the desire to support the restrictive underlying platform. Finally, C code is written for this algorithm, with external references to SensorWare functions. The compiled native code has a size of 764 bytes (without including the size of SensorWare functions called from within the native code).

B. *Porting SensorWare to a platform*

In this subsection we will present some of the issues while porting SensorWare to a platform. We consider our iPAQ-based prototype as the testbed. A full description of the platform can be found in section VI.A. Here it is sufficient to know that the node has one radio and one sensing device, and that the underlying OS is Linux.

First, we should add the proper capabilities to SensorWare by creating a virtual device for the sensing device (the radio has a virtual device by default). This means name and register the device by calling the function:

```
create_device(char* name, int (*query)(), int (*act)(), void* (*createEventID)(), int (*disposeEventID)(), void* (*task)() )
```

As it can be seen by the declaration of the create_device function we need to define the four functions to parse the arguments of the four standard interface commands, plus a function to be executed by the thread/ task of the device. Not going any further into the definition of these functions, we are sufficed to say that they are very similar to the radio device functions.

The next step is to define the OS-specific code. More precisely, have the ability to create threads and use mailboxes/queues. For the definition and creation of threads we use the pthreads (i.e., posix threads) provided by Linux. Even though mailboxes are available in Linux, we chose to construct our own structures using semaphores. Finally, the hardware-specific code is directly provided by the Linux's device drivers.

VI. IMPLEMENTATION

Some active sensor frameworks choose to evaluate their performance by showing their expressiveness. They create a distributed algorithm for a particular application and compare it against a more centralized approach (usually a distributed database approach). We believe that the energy savings from such comparisons are evident for *any* active sensor framework and do not add value to the investigation and evaluation of the framework. To evaluate SensorWare we chose to implement it and measure the overheads we are paying for dynamic programmability. How much memory do SensorWare and its components occupy? How much delay is introduced by various SensorWare operations? How much slower and consequently how much more energy-consuming is SensorWare compared to native code approaches? These questions are answered in the following subsections. We begin by a description of the implementation platform.

A. *Platform description*

The prototype platform used in the implementation and evaluation of SensorWare was built around the iPAQ 3670 [16]. The iPAQ has an Intel StrongARM 1110 rev 8 32 bit RISC processor, running at 206Mhz. The flash memory size is 16Mbytes and the RAM memory size 64Mbytes. The OS installed is a familiar v0.5 Linux StrongARM port [9], kernel version 2.4.18-rmk3. The compiler used, is the gcc cross-compiler. A wavelan card [30] is used as the radio device and a Honeywell HMR-2300 Magnetometer [14] as the sensing device.

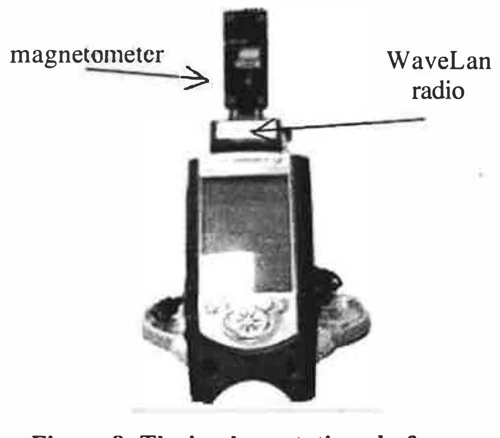
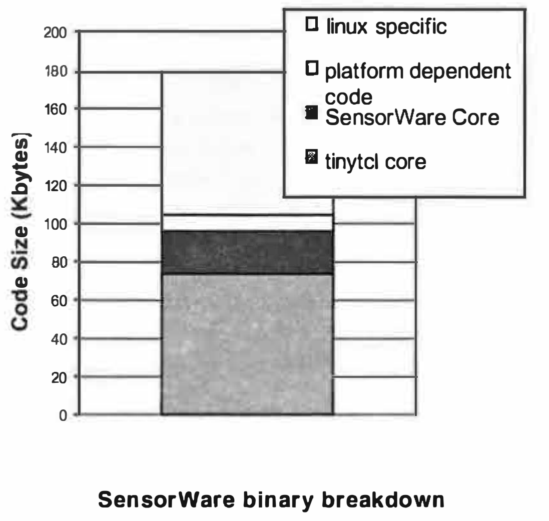


Figure 8: The implementation platform

SensorWare is also ported into the Rockwell WINS nodes [26] that also have a StrongARM processor, but only 1Mbyte of flash memory. Both eCos [6] and microC/OS-II [19] were used as operating systems for these nodes.

B. Memory size measurements

The first question to answer is how much size does the whole framework occupy. Figure 9 shows that the total size is 179Kbytes and it is consisted of 74Kbytes of Linux specific code (e.g., kernel, libraries), 74Kbytes of a stripped down Tcl core called tinyTcl, 22Kbytes of SensorWare code and 8Kbytes of platform dependent code (i.e., functions to access the hardware). The bottom part of the figure shows the breakdown of the SensorWare core part into smaller parts.



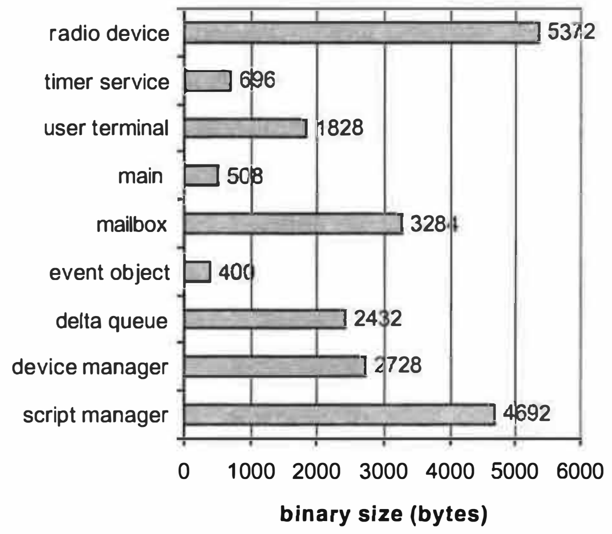
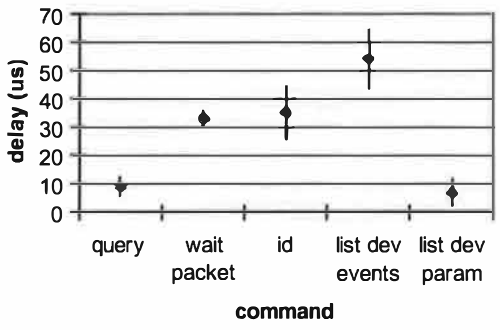
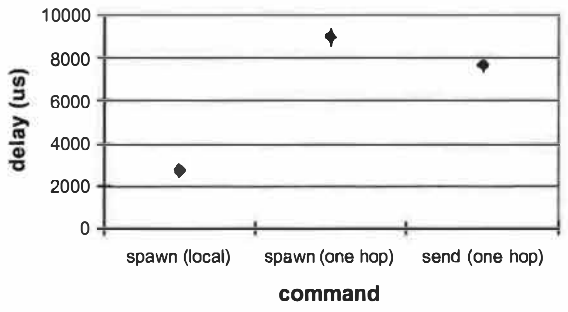


Figure 9: Code size breakdown

C. Delay measurements

The next question to answer is how long do different basic commands need to execute. We measured each command individually 100 times under the same basic conditions (only one script executing) and derived an average and standard deviation for the delay. Most commands exhibited negligible variance. All the commands, except the ones that used the radio and the one that spawned a 50byte script, have an execution time less than 0.3msec.





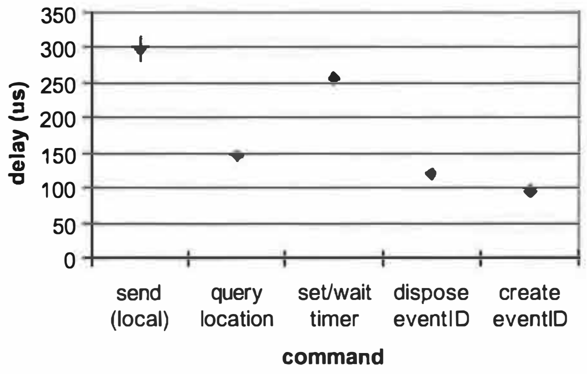


Figure 10: Execution times of SensorWare commands

The top graph of the figure 10, shows commands with less than 0.06msec delay. The last two commands that return some part of the device's state are internal to SensorWare and not exported for script use. The middle graph shows the most time consuming commands. The first one spawns a 50 byte script locally. The other two commands use the radio to spawn a script in a neighboring node and send a message in a neighboring node. The delay for achieve these two operations is dominated by the radio transmission time. Note that the send command and some operation modes of the spawn command, do not wait for the whole operation to finish, instead they return as soon as they hand off the task to the radio device. In the graph, the total operation time is shown. The bottom graph of figure 10, shows yet another set of delays. Of particular interest is the set/wait timer delay. For this instance, we measure the delay to set a zero-valued timer and wait for its expiration. In essence we are measuring the overhead of real-time measurements in scripts. The overhead is 0.25msec with very small variation, which means that the overhead is virtually constant. Therefore, we can internally subtract this number each time a script sets a timer, in order to measure the true desired time.

In order to acquire all delay measurements we used the gettimeofday() system call. This function is based on the timer count register found in the StrongARM processor. The accuracy of this method is measured to be 1μsec.

D. *Energy measurements and related tradeoffs*

Finally, we are interested in knowing the energy overhead from the interpreted nature of SensorWare. For that purpose we compare the interpreted version of the script presented in section V.A., with a compiled native code version of the same algorithm. The native version uses the services that SensorWare provides by directly calling the appropriate functions. Since most of the work inside a script is done by the SensorWare commands and services (which are implemented in native code) we do not expect a significant change when we resort to fully native code. Indeed, we measured an 8% speedup of the native code compared to the interpreted code. We acquired this number by measuring the total execution times of both codes, and *excluding* time periods when the code was accessing the radio, or was waiting for events to occur. Essentially, the time we measured, was the non-idle CPU time. This time is linearly coupled with the energy spent on the CPU, assuming that we have a mechanism to shut down the CPU during idle time. Thus a reduction of 8% in the non-idle time, directly translates to a reduction of 8% in CPU-energy spent.

As we already mentioned in section V.A, the script has a final compressed size of 209 bytes, while the native code has a size of 764 bytes. So even if the native version executes faster (and potentially consumes less energy, by allowing to shut down the CPU during idle time), there is an energy overhead related to its transmission. The wavelan radio in typical operation would spent 0.47mJ to transmit the script, and 1.10mJ to transmit the native code (including the MAC overhead). Thus, the energy difference between the two transmissions is 0.63mJ. The typical power for the StrongARM is 230mW, so 0.63mJ are spent in 2.7msec. From these numbers we deduce that if the native code uses StrongARM for 2.7msec less than the interpreted code then its initial transmission energy overhead is balanced. For the particular algorithm that we tested, 8% speedup is translated into 1.2msec gain in absolute numbers. So for the particular algorithm and hardware platform, transmitting and executing native code is not beneficial overall. For applications with heavier computation workload it might be desirable, from an energy viewpoint, to transmit and execute native code. Also in other platforms the tradeoff points might change as the CPU and radio characteristics change. Although usually, in low-end nodes, the CPU is slower and the radio much slower than our platform making the communication costs more dominant and thus favoring the script approach. Finally, a native-code approach would sacrifice the portability of the code in several platforms, and most importantly would sacrifice the code safety offered by the scripts (refer to [2] for more information on scripts code safety).

VII. CONCLUSIONS

In this paper we argue that the development of a framework based on a scripting abstraction where the scripts are mobile, will help bring many desired properties in sensor networks. It will make the sensor networks programmable and open to external users and systems, keeping at the same time the efficiency that distributed proactive algorithms have. We explain the framework's architecture and present code examples. Through our implementation we are able to measure the time and energy overheads that we are paying for programmability and explore some part of the solution space for sensor node run-time environment abstractions.

VIII. REFERENCES

[1] P. Bonnet, J. Gehrke, and P. Seshadri, " Querying the Physical World", IEEE Personal Communications, October 2000.

[2] A. Boulis and M. B. Srivastava, " A Framework for Efficient and Programmable Sensor Networks", In Proceedings of: OPENARCH 2002, New York, NY, June 2000.

[3] A. Boulis, "Illustrating Distributed Algorithms for Sensor Networks", http://www.ee.ucla.edu/~boulis/phd/Illustrations.html

[4] A. Boulis and M. B. Srivastava, "Node-level Energy Management for Sensor Networks in the Presence of Multiple Applications," The first IEEE Annual Conference on Pervasive Computing and Communications (PerCom 2003), Dallas-Fort Worth, TX, March 23-26, 2003.

[5] L. Clare, G. Pottie, J.R. Agre, "Self-Organizing Distributed Sensor Networks", Proceedings of SPIE conference on Unattended Ground Sensor Technologies and Applications, pp. 229-237, April 1999.

[6] eCos: Embedded Configurable Operating System, http://sources.redhat.com/ecos/

[7] D.Estrin, R.Govindan, J.Heidemann (Editors), "Embedding the Internet", Communications of the ACM. Vol. 43, no 5, pp. 38-41, May 2000.

[8] D. Estrin, R. Govindan, J. Heidemann, S. Kumar, "Next Century Challenges: Scalable Coordination in Sensor Networks", ACM Mobicom Conference, Seattle, WA, August 1999.

[9] Familiar Project, "http://familiar.handhelds.org".

[10] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govidan, D. Estrin, D. Ganesan, "Building Efficient Wireless Sensor Networks with Low-Level Naming", Proceedings of Symposium of Operating Systems Principles, October 2001

[11] M. Hicks, P. Kakkar, J. Moore, C. Gunter and S. Nettles, "PLAN: A Packet Language for Active Networks", Proceedings of the International Conference on Functional Programming (ICFP'98), 1998.

[12] J. Hill and D. Culler, "A wireless embedded sensor architecture for system-level optimization", Intel Research IRB-TR-02-00N, 2002.

[13] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, "System Architecture Directions for Networked Sensors", Proceedings of ASPLOS-IX, November 2000 Cambridge, MA, USA.

[14] Honeywell HMR-2300 Magnetometer, http://www.ssec.honeywell.com.

[15] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks", MobiCOM '00, August 2000, Boston, MA.

[16] iPAQ 3670, http://thenew.hp.com/.

[17] C. Jaikaeo, C. Srisathapornphat, and C. Shen, "Querying and Tasking of Sensor Networks", SPIE's 14th Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Control (Digitization of the Battlespace V), Orlando, Florida, April 26-27, 2000.

[18] D. Kotz, R. Gray, "Mobile Agents and the Future of the Internet", in ACM Operating Systems Review, 33(3), 1999.

[19] J. Labrosse, " MicroC/OS-II: The Real Time Kernel", CMP Books, November 1998.

[20] P. Levis, D. Culler, "Maté: A Tiny Virtual Machine for Sensor Networks." Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X), October 5-9 2002.

[21] S. R. Madden, R. Szewczyk, M. J. Franklin and D. Culler, Supporting Aggregate Queries Over Ad-Hoc Wireless Sensor Networks, Workshop on Mobile Computing and Systems Applications, 2002.

[22] J. K. Ousterhout, "Scripting: higher level programming for the 21st Century", Computer, vol.31, (no.3), IEEE Comput. Soc, March 1998. p.23-30.

[23] J. K. Ousterhout, "Tcl and the Tk toolkit", Addison-Wesley, 1994.

[24] G.J. Pottie and W.J. Kaiser, "Wireless Integrated Network Sensors", Communications of the ACM. Vol. 43, no 5. May 2000.

[25] Reactive Sensor Networks, http://strange.arl.psu.edu/ RSN/

[26] Rockwell WINS nodes, http://wins.rsc.rockwell.com/

[27] SenseIT program, http://www.darpa.mil/ito/research/sensit/index.html

[28] C. Srisathapornphat, C. Jaikaeo, and C. Shen, "Sensor Information Networking Architecture", International Workshop on Pervasive Computing (IWPC'00), Toronto, Canada, August 21-24, 2000.

[29] D. Tennenhouse, "Proactive Computing", Communications of the ACM. Vol. 43, no 5, pp.43-50, May 2000.

[30] Wavelan card, http://www.orinocowireless.com

[31] V. Wen, A.Perig, R. Szewczyk, "SPINS: Security suite for Sensor Networks", Proceedings of MOBICOM'01, Rome, Italy, July 16-21, 2001

IX. APPENDIX

A. *The SensorWare Language*

SensorWare supports Tcl syntax and the following 41 Tcl commands: append, array, break, case, catch, concat, continue, error, eval, expr, for, foreach, format, global, if, incr, info, join, lappend, lindex, linsert, list, llength, lrange, lreplace, lsearch, lsort, proc, regexp, regsub, rename, return, scan, set, split, string, trace, unset, uplevel, upvar, while.

There are 11 other commands defined by SensorWare that essentially abstract the node's run-time environment. They are:

```
spawn [ -[f] [d] [p] [m] [rc] [rs] [ru] ] [<node_list>] <code>
        [<variable_list>]

replicate [ -[f] [d] [p] [m] [rc] [rs] [ru] ] [<node_list>]
        [<variables_list>]

migrate [ -[f] [d] [p] [m] [rc] [rs] [ru] ] [<node_list>]
        [<variables_list>]

send    (<node_id>|*)[:<script_name>[:<user_id>[:<app_id>]]]
        <message>

setTimer <timer_name> <value>

disposeTimer <timer_name>

query <device_name> [ var_arg... ]

act <device_name> [ var_arg... ]

createEventID <device_name> <eventID> [ var_arg... ]

disposeEventID <device_name> <eventID>

wait <event_name>...
```

Legend: [ ] indicates optional, < > indicates a variable (either a Tcl variable or an SensorWare variable such as an eventID or a timer name), the suffix "_list" in variable names indicates that the variable is a list (i.e., zero or more elements). The symbol "var_arg ..." indicates variable arguments. The modifier "..." indicates a list of arguments of the preceding argument type.

There are 6 reserved Tcl variable names. These are: parent, neighbors, event_name, event_data, msg_sender, msg_body.

There are 7 reserved words used as arguments in some commands. By reserving words for commonly used features we compact the scripts further. These are: anyRadioPck, anyTimer, add_user, sensor, value, radio, timer.

An Entity Maintenance and Connection Service for Sensor Networks

Brian Blum, Prashant Nagaraddi, Anthony Wood, Tarek Abdelzaher, Sang Son, Jack Stankovic
Department of Computer Science
University of Virginia, Charlottesville, VA 22904

Abstract

In this paper, we present a middleware architecture for coordination services in sensor networks that facilitates interaction between groups of sensors which monitor different environmental events. It sits on top of the native routing infrastructure and exports the abstraction of mobile communication endpoints maintained at the locations of such events. A single logical destination is created and maintained for every environmental event of interest. Such destinations are uniquely labeled and can be used for communication by application-level algorithms for coordination and sensory data management between the different event locales. For example, they may facilitate coordination, in a distributed intrusion scenario, among nodes in the vicinity of the intruders.

We evaluate our middleware architecture using GloMoSim, a wireless network simulator. Our results illustrate the success of our architecture in maintaining event-related communication endpoints. We provide an analysis of how architectural and network dependent parameters affect our performance. Additionally we provide a proof of concept implementation on a real sensor network testbed (Berkeley's MICA Motes).

1 Introduction

The impending proliferation of sensor networks calls for new services and middleware for distributed deeply embedded computing. We consider ad hoc networks formed by dropping wireless computationally-equipped sensors (e.g., from an airplane) onto a dangerous or inaccessible infrastructureless environment such as a disaster area or a hostile territory behind enemy lines. The lack of an infrastructure implies that no workstations or other centralized computing equipment are present for management or information processing purposes. Such sensor networks will therefore have to host their own distributed embedded computation. They will execute their mission autonomously while interacting with spatially distributed external events in the physical environment. In this paper, an environmental event refers to an ongoing activity, such as the motion or presence of a vehicle, that persists in the physical world for some continuous interval of time.

A new distributed computing paradigm is needed to support the writing and execution of distribution applications for such networks. Due to the tight coupling between computation in the sensor network and events in the environment, one key requirement of such a paradigm is to support applications that coordinate teams of sensor and actuator nodes in the vicinity of different external events of interest. For example, one may want to exchange data among all nodes in the vicinity of hostile targets in the sensor field to determine a plan of attack.

The aforementioned coordination problem offers an interesting research challenge to the communication subsystem pertaining to mobility. In PDAs and Wireless LANs, supporting mobility typically refers to maintaining connectivity between individual devices despite their changing spatial relationship to one another. In contrast, what moves in our scenario are the external environmental events. Sensor network nodes themselves remain relatively motionless. In this paper, we present middleware that allows programmers to think of mobile external events in terms of abstract persistent entities that logically form in the network as a response to appropriate sensor readings. The middleware forms and maintains a unique entity around each event. These entities are addressable and act as communication destinations (end-points). Note that we define an *event* as something in the environment that causes a sensor to report a certain reading (e.g., fire, moving vehicle, etc.) and an *entity* as the abstract addressable equivalent of this event as referenced by the programmer.

The communication problem is therefore to maintain the abstraction of transport-layer connections between different entities, when each entity is composed of a changing set of sensor nodes at the location of a mobile external event. Such mapping is made complicated by several factors. One is the need for seamless end-point migration across nodes as the event moves. Another is that sensor nodes that become aware of an external event should be able to decide whether it is the same event previously seen by other sensors or a new event. Otherwise, an incorrect event list will be collectively maintained or an incorrect mapping will result between events and communication endpoints. This paper reports a communication architecture which resolves these challenges.

Our middleware hides the details of sensor group formation around environmental events, end-to-end connection establishment between different entities, and entity maintenance to ensure that a single abstract entity is created and maintained for every event of interest in the environment. This architecture's ability to ensure a one to one relationship between abstract entities and environmental events simplifies communication, facilitates coordination, and reduces programming complexity by providing communication with persistent entities instead of dynamically changing sets of individual nodes.

Our techniques are geared for the case where events are reasonably sparse (i.e., the tracked environmental targets are generally not in close proximity). Disambiguating nearby targets is an inherently difficult problem that is not addressed in this paper. The reader may want to think of our architecture as imposing a resolution constraint. Targets that are closer than the available resolution cannot be individually distinguished. As we shall see later, the resolution is of the order of the communication radius of the sensor nodes. In the physical sensor node prototype available to the authors this radius can be adjusted from several inches to hundreds of feet.

The remainder of this paper is organized as follows. We review related work in Section 2. An overview of entity establishment and maintenance as well as details of the underlying protocols follows in Section 3. Detailed simulation results are discussed in Section 4. A description of an implemented proof-of-concept prototype is given in Section 5. Finally, we explore future work and conclude in Section 6.

2 Related Work

Sensor networks [11] have recently emerged as a promising platform for a myriad of distributed embedded applications in defense [42] and scientific exploration [21]. A typical sensor network is highly distributed and composed of thousands to hundreds of thousands of individual nodes. Communication protocols are therefore a very important research topic in sensor networks.

Most prior work on communication in sensor networks has focused on the lower layers in the protocol stack. For example, [41, 45] propose MAC layer protocols designed for sensor networks. At the network layer, protocols such as DSDV [27], DSR [14], AODV [29] and TORA [25] have gained popularity as routing solutions for ad hoc wireless networks. These protocols are designed for networks with identifier-based node addressing. Recent sensor network research suggests alternative addressing schemes that do not rely on having destinations with specific identities. Instead, it has been proposed that routing in sensor networks be attribute-based where the destination is reached by its attributes such as location or sensor measurements. For example, LAR [17] and DREAM [3] propose location-aware routing protocols, where the destination is implicitly defined by its physical location. Directed diffusion [13] and the intentional naming system [1] provide routing and addressing based on data interests. A related effort is attribute-based naming [32], proposed for an Internet environment, which allows queries to be routed depending on the requested content rather than on the identity of the target machine. Our work falls in the general category of attribute-based communication. We provide an infrastructure where communication end-points are placed at the locations of specific events in the environment. Unlike prior work on attribute-based addressing, we focus on protocol dynamics that arise due to the motion of such events in the external world. We aim to maintain the persistence and uniqueness of these communication end-points as the event moves and discuss factors that affect the maximum trackable speed of these events. Also, unlike routing-layer approaches, our architecture sits in the transport layer on top of geographic forwarding.

Mobile end-points have been addressed in traditional and ad hoc computer networks. For example, Energy-aware routing protocols were proposed such as Span [6] and GAF [43] for communication between mobile nodes. In Mobile IP [26], mobile hosts are free to migrate between LAN's while they remain connected by a home agent residing at their home address. Another mechanism for maintaining mobile connections uses DNS to provide the indirection necessary to support mobility [34]. We address mobility in a different sense in our protocol. Whereas the aforementioned protocols assume moving nodes and provide for communication between these moving nodes, we assume static nodes but moving events (and thus entities) in our system and provide a migratory end-point infrastructure for communication between these moving entities. A recent protocol, called TTDD [44], addresses communication between moving sources and sinks in a sensor network. Our work differs from this in the fact that we provide for the creation and maintenance of abstract entities to facilitate communication between moving events in the network. Also, our communication end-points are bi-directional as opposed to being statically designated as sources or sinks.

Several algorithms exist that provide clustering and various granular levels of group formation on both the network and application layers. The (α, t) framework [22, 37], and Landmark Hierarchy [39] organize nodes into hierarchical groups as a solution to routing. LEACH [10], ASCENT

[5], SPAN [6], and GAF [43] form groups or share data locally to conserve energy and power down unused or unneeded nodes. The AC Hierarchy [7, 8, 38] forms hierarchical clusters covering the entire sensor network and provides a high level programming abstraction for division or simplification of the sensor network. Finally, GLS [19] and MASH [33] provide cluster-based location or query services for locating data or nodes. Unlike these cluster-based or group-based algorithms we provide the abstraction of tracking groups linked directly to environmental events of interest. Although some of these algorithms such as GLS could be used in parallel with our work for object lookup, and although various ideas from these protocols are similar or could be used to enhance the efficiency or functionality of our modules, none provide sufficient support for entity formation around environmental events of interest and end-to-end connection establishment and maintenance between moving entities.

Our work is also complementary to several research efforts that aim to provide new abstractions and paradigms for distributed computing in sensor networks. For example, MagnetOS [2], exports the illusion of a single Java virtual machine on top of a distributed sensor network. The application programmer writes a single Java program. The runtime system is responsible for code partitioning, placement, and automatic migration such that total energy consumption is minimized. Mate [18] is another example of a virtual machine developed for sensor networks. It implements its own bytecode interpreter, built on top of TinyOS [11]. The interpreter provides high-level instructions (such as an atomic message send) which the machine can interpret and execute.

To the authors' knowledge, the communication architecture proposed in this paper is the first that provides middleware support to tracking applications for group formation around environmental events, end-to-end connection establishment between different entities, and abstract entity maintenance to ensure that a single entity is formed and maintained for every event in the environment. The architecture ensures a one to one relationship between abstract entities and environmental events thus simplifying communication. It reduces programming complexity by allowing communication with entities rather than individual nodes.

3 Service Architecture

The ability of a sensor network to closely interact with the environment in which it has been deployed gives rise to a multitude of applications in which code execution is tightly linked to the locations of environmental events. Appropriate communication abstractions are required to isolate distributed application programmers from accounting for the changing locations of environmental events in the vicinity of which the communication end-points of their applications are located. Our architecture provides such abstractions by a combination of (i) a team management framework for maintaining proper mapping between communication endpoints and external events (ii) a transport layer protocol for communication among event-related endpoints. Together, they maintain communication end-points associated with mobile external events, as well as maintain connectivity among such endpoints. This architecture is independent of the underlying radio, data link, and network layer protocols making it applicable in principle to an array of sensor network platforms.

Such programming abstractions are desired in applications where one wishes to interact with physical events in the environment that, for one reason or another, do not communicate directly with the network. By forming an abstract entity that moves with the event, we can associate state and behavior with the physical event. For a hostile target being tracked, this state and behavior could include monitoring the number of shots fired from a tank or the distance an object has traveled.

3.1 The Entity Communication Problem

The problem addressed by our architecture is more formally described as follows. We consider a dynamically changing set S of events in the physical environment of the sensor network. Let the physical location of each event $E_i \in S$ at time t be denoted $L_i(t)$. A node is said to be in the *vicinity* of event E_i at time t if it is within sensor range of the event's location, $L_i(t)$. In this paper, we assume that environmental events are localized. In other words, their location is described by a single point in space, as opposed to an area. This definition applies to tracking vehicles, finding survivors, monitoring wild animals, or detecting localized fires. It does not apply to applications involving distributed phenomena such as detection of large chemical spills. We assume that events can be detected independently by individual nodes in the sensor network based on their local measurements. For example, detecting a magnetic signature in a desert battle area would usually be indicative of a passing armored vehicle. Finally, we assume that events are sparse. In other words, the signatures of different targets are generally not overlapping.

Let T_i denote the set of nodes in the vicinity of event E_i. The objective of our architecture is to maintain a *unique* addressable destination associated with each event E_i, such that sending data to this logical event address causes delivery of this data to T_i regardless of the location $L_i(t)$. In the current implementation, we elect a leader out of set T_i. The leader, among other things, is responsible for communication with remote destinations. Hence, in the above problem statement, we define delivery of a message to T_i as delivery of the message to the current entity leader who by definition belongs to the set T_i. What the leader does with the message is an orthogonal issue in our architecture.

Note that once the aforementioned addressing and communication problem is solved, it becomes trivial to associate multiple communication end-points with each entity simply by demultiplexing the received message based on a port number in the message header, in the same sense that UDP creates multiple ports over IP.

3.2 Sensor Network Assumptions

Our underlying sensor network typically consists of thousands of small sensor nodes thrown arbitrarily (e.g., from the air) onto a large target area, such as a battlefield or the scene of a natural disaster. Individual nodes have resource limitations associated with small physical size including low-power batteries, relatively slow processors, and limited memories. They are capable of wireless communication and once deployed form a large-scale ad hoc network. A key assumption is that the formed sensor network has no pre-existent infrastructure or centralized services. It is precisely the difficulty of creating such an infrastructure in harsh or inaccessible environments that motivates the sensor network approach. An example of computationally equipped wireless sensor devices that meet the above description is the MICA mote [12], which we use in our experimental prototype.

Once deployed, nodes in a sensor network are assumed to establish their location and remain motionless except due to environmental factors such as wind and water. In an important departure from the typical mobile ad hoc wireless network model, nodes in sensor network literature do not have IP-address, and do not run the TCP/IP protocol suite. Instead of possessing unique ID's, sensor network nodes are usually referenced by attributes such as location. Both localization services [4, 30] that establish sensor network coordinate frameworks, and location-based routing services [17, 3] that route messages geographically have been discussed at length in previous literature.

Sensor nodes may perform local processing as appropriate for their particular application. This could be aggregating and reporting raw data, triangulating the position of an event, coming to agreement about an actuation or reporting strategy, or performing distributed event analysis. A distributed application, such as a distributed intrusion response system, may need to pass the results of such local processing among the respective groups of sensors to coordinate a sensor network reaction.

Our service can be thought of as a distributed protocol that sits in the transport layer of the sensor node's protocol stack. In its basic form, the protocol implementation consists of two modules, namely, the *entity management module* (EMM), and the *entity connection module* (ECM). These modules are shown in Figure 1. As the name suggests, the EMM forms a local entity in response to sensor readings at the locations of environmental events. It maintains the unique identity of this entity as the event of interest migrates in the environment. The ECM provides a means for entity registration, maintains communication end-points, and provides connectivity to allow communication among different entities. The following sections discuss the details of the APIs and implementations of the aforementioned modules.

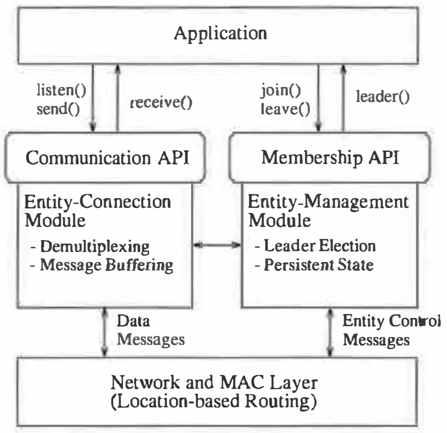


Figure 1. Service Architecture

3.3 Entity Management Module

The entity management module (EMM) provides an entity formation and maintenance service. The EMM has several essential functions. First, an entity must be created and identified when an event first occurs in the sensor network. Second, once established, the EMM must maintain this single entity and prevent spurious entities from forming around a previously abstracted event. While an entity exists the EMM must maintain its persistent state such as its unique identity which signifies the local environmental event. Finally, the EMM is responsible for ensuring that nodes that sense an event for the first time know when to become a member of an existing entity and when to spawn a new entity around the sensed event. These functions are described below.

3.3.1 Functional Overview

At any given time, multiple events in the environment give rise to multiple entities which host the communication end-points in the sensor network. An entity is a set of nodes in the vicinity of a single environmental event. We call nodes whose sensors detect the signature of the corresponding event, *entity members*. These nodes are said to be within

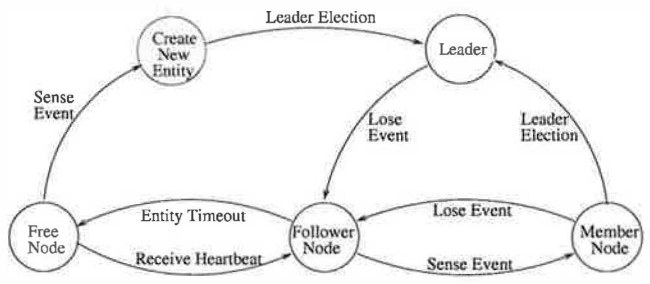


Figure 2. Node state transition diagram

the *sensory horizon* of the target event. To the rest of the sensor network, an entity acts as a single whole. The fact that the entity may consist of multiple nodes is hidden and the identities of these nodes are abstracted away.

Entity members volunteer to be an *entity leader*, a central node responsible for communication and group maintenance, as well as running entity-specific application code. Once the entity leader no longer detects the event, it hands-off leadership (and current state) to another node by sending a *relinquish leadership* message. As the event migrates or expands, the leadership handoff mechanism built into the EMM ensures that the entity and its stored application state migrate with it. The application on a node is informed when the node becomes a leader so that it can pick up computation from where the previous leader left it. The application is also notified when the node ceases to be leader.

A key requirement is to ensure that only one entity is spawned for the same environmental event. The EMM protocol achieves this requirement by announcing the existence of an entity to nearby nodes within a distance called the *awareness horizon*. By design, the awareness horizon is larger than the sensory horizon. Nodes in the awareness horizon that cannot sense the event are called *entity followers*, as distinguished from entity members. These nodes are prevented from spawning new entities.

Nodes join the awareness horizon upon the reception of a bounded-hop broadcast (*heartbeat*) from the nearby EMM leader. Upon receiving a heartbeat, a node sets a corresponding *entity timeout timer*. The node ceases to be a follower when the timer expires. The timer is re-initialized upon the receipt of each new heartbeat. If the node senses the event signature before the entity timeout timer expires it becomes a member of the existing entity. If the timer had already expired the node is no longer a follower and will create a new entity. This mechanism prevents multiple entities from being spawned for the same environmental event. Figure 2 depicts the node state transition diagram between follower, member, and leader states, as well as the free state in which a node is not cognizant of any entities. For readability, the only transitions shown are those between different states (as opposed to loop-backs from a state to itself).

Note that, while inhibiting entity followers from creating new entities is essential for preventing redundant representation of the same target, it has one unfortunate side effect. Namely, the mechanism may prevent nodes from reporting secondary events in the vicinity of events already reported. Since we assume that events are sparse, the hope is that cases like the above are not common.

3.3.2 Entity Uniqueness

Entity uniqueness is the algorithm property that states that only one entity is associated with any given environmental event. In this section, we take a closer look at the conditions under which this property holds true. In general, there are two cases in which entity uniqueness can be compromised. The first case occurs at excessive target speeds. If the target moves in the environment fast enough, far apart nodes can detect it at about the same time and create independent entities to represent it. The second case occurs due to message loss or node failures which may prevent proper leadership handoff. Consequently a new leader may emerge that does not inherit the right entity identity from the old leader, causing a different entity to emerge for the same environmental event. In the following, we quantify the maximum event speed that preserves entity uniqueness and discuss provisions to ensure robustness in the face of failures.

Event Speed: The key rule which inhibits creation of duplicate entities is that followers of existing entities cannot spawn new entities. Instead, when they eventually sense the event, they must join the membership of the entity of which they were followers. By extending awareness of the event (i.e., the awareness horizon) beyond its sensory horizon we can ensure that new nodes will always become aware of the current entity before they sense the event. Hence, a single unique entity will exist for each event in the environment. The above uniqueness property is violated only if the event moves fast enough in the environment such that it is sensed by nodes outside of the awareness horizon before information of this event is propagated to them. Controlling the awareness horizon therefore determines the maximum tolerable event velocity as will be detailed below.

Note that a new leader is elected once the old one stops sensing the target. This new leader will cause the center of the awareness horizon to shift to its new location. If leader re-election and heartbeat propagation took zero time, the system could theoretically track infinitely fast targets as long as the awareness horizon was at least double the sensory horizon. This is because the current leader would always be within sensor radius from the target and no other node within the sensory horizon could be more than twice the sensor radius away from the leader. Hence, all nodes who sense the target are always within the awareness horizon and are therefore inhibited from creating new entities. In reality, however, leader re-election and heartbeat propagation take time. If the maximum combined leader re-election and heartbeat propagation delay was D, it is easy to show that the maximum speed that preserves entity uniqueness is $(awareness\ horizon\ -\ 2 \cdot sensory\ horizon)/D$. It should be noted that the above is a conservative estimate. Entity uniqueness will not be compromised immediately at higher target speeds.

Robustness to Message Loss and Failure: To prevent handoff failure in the case that an entity leader dies or otherwise fails to send out the relinquish heartbeat message, each entity member sets a *failed leader timer*. This timer, upon expiration, prompts an entity member to assume the entity leader role and begin sending heartbeats after an additional random delay (to prevent simultaneous takeover collisions). This failed leader timer must be set to a value larger than the heartbeat period, the interval between heartbeats, to ensure that timer expiration does not occur prematurely while the current leader is still alive. Depending on expected message loss, one might also set this timer to a value greater than two or three times the heartbeat period to prevent inopportune and premature handoff when heartbeats are lost or subject to collisions. Note the delay that a node waits before assuming the entity leader role could be determined in accordance with the strength of a node's sensor reading, whether or not this sensor reading is growing or shrinking in strength, the number of entity members that are direct neighbors of that node, or by some other appropriate metric.

Message loss can also prevent nodes within the awareness horizon from getting the leader's heartbeats. Consequently, these nodes may not become aware of the entity and may create a spurious one when they sense the event. To kill such spurious entities, we employ a mechanism that associates larger weights with older entities and biases nodes against joining entities with smaller weights. The mechanism maintains an *alive counter* at the leader of each entity. This counter is propagated through heartbeats and its value is accumulated across leader handoffs. When a new entity is first created, its counter is initialized to 0. This value is then incremented for each heartbeat sent out and is therefore a reflection of how long the entity has remained in the network. When a node tries to spawn a new entity, every neighbor that is already part of an entity with a higher alive counter ignores the new node. Hence, the faulty node is isolated. The mechanism will send a kill message to the faulty node to request termination of its spurious entity.

The above mechanism serves to prevent spurious groups from forming in the presence of message loss, but fails to handle the case where events of the same signature migrate across one another's path. To handle this more complex scenario we define a compile time specified threshold, *min time alive*, to ensure entities that have existed over some time period remain after crossing paths with an even older entity. When a node of entity $E1$ receives a heartbeat from the leader of another entity $E2$ and both entities have an alive counter set greater than the min time alive threshold, we require that both entities coexist. In this case, nodes independently apply the EMM protocol with respect to each entity. They may be within the awareness horizon of multiple entities at the same time. When they sense the event, they become members of all entities that exceed the min time alive threshold of which they are aware.

3.4 Entity Management API

The API exported by the EMM consists of *join(signature)* and *leave(signature)* primitives which the application calls when it first senses or stops sensing an event signature respectively. The code of the signature is passed as the input parameter to these primitives. For example, if the magnetic signature of a vehicle is sensed, the node calls *join(Magnetic)*. Unlike traditional group communication, the *join()* does not take a group identifier as input. Instead, it returns as output the identity of the environmental event the application just sensed (i.e., the identity of the entity for which the node was a follower at the time *join()* was called). If the node is a follower of multiple nodes, a list of identities is returned. The semantics are that the node has joined the respective list of entities. If a node is not a follower of any entity, a new entity is created when *join()* is called and the code of the new entity is returned.

The EMM also requires the application to implement a handler for an upcall called *leader(entity,on_off)*. The upcall contains an entity id as a parameter as well as a boolean that tells the application that its node has just become or ceased to be leader of the named entity. Application code would typically check whether it is the leader or a member, and execute the corresponding part of its typically distributed data processing algorithm based on the assigned role.

The last part of the API is a *store(entity,state)* and *get(entity,state)* call that allows the application on the leader node to save and retrieve persistent state of a named entity. The entity name is passed as input parameter to the call. Typically, the leader would save its state after each iteration. This state is transmitted in the EMM heartbeats to all members of the entity. Upon leader handoff, the new leader would use the above API to get the most recent previously communicated state. It would then resume the iterative application from that point onwards.

3.5 Entity Connection Module

The entity connection module (ECM) provides a basic end-to-end location and communication service between mobile entities. ECM is therefore the equivalent of UDP for sensor networks, with the exception that destinations are migratory entity leaders, not IP hosts. An application can utilize the ECM's API to communicate messages to and from logical entities without concern for where that entity resides, how it is maintained, or what particular nodes compose it. An application programmer can therefore initiate queries and interact with environmental events that migrate throughout the sensor network.

The ECM exports a subset of a socket-like API. All applications are assumed to have well-known ports. In the current protocol 256 ports are supported based on a byte in the message header. We do not support dynamic application-to-port binding. This is because in our target platform, namely, Berkeley's MICA motes running TinyOS, applications are structured as a graph of permanently wired modules. The ECM demultiplexes incoming messages as upcalls to different application modules depending on their port number. The association of port numbers and upper layer modules is defined in a compile-time configuration. At run-time an application can call *listen()* to notify the ECM that it is ready to receive messages on its assigned port. Subsequently, the ECM propagates messages on this port to the application. If a message arrives for a port on which no application is listening, the message is dropped.

Connections are identified by a *<Entity ID, Port Num>* pair. When an entity is spawned, entity registration is invoked by the ECM. This registration utilizes a directory service similar to the indirection infrastructure described in [35]. Namely, each entity maintains replicated pointers to its current location in a region of the sensor network determined by a hash function. The hash key is the signature identifier associated with the entity. By hashing this key, the ECM can determine the location of the directory region associated with a particular type of environmental event, then query the directory for all entities that are currently following events of that type. Queries to this directory service supply entity leader information pertinent to establishing mobile connections.

When connecting with an entity, the ECM looks up the last known entity leader based on the *<Entity ID, Port Num>* pair provided in the application call. If this information is older than a specified threshold the directory service is contacted for updated information. The returned last-known leader is used as a connection point for communication. Upon receiving a message, an endpoint updates its table of last-known leaders with that contained in the header. The more traffic exchanged between the endpoints, the more up-to-date the leader information is.

Leadership information is retained in the ECM in a limited-size table. When the table is full, replacement is done on a least-recently-used basis. The ECM of an entity periodically refreshes the directory region, at an interval called the *directory refresh rate* to ensure that its information remains up to date. In addition, past followers of an entity remember the location of the last known leader for a time interval that exceeds the directory refresh rate. Hence, messages sent to the old location of an entity are forwarded to the current location when they intercept the entity's trail.

4 Simulation

To fully understand and validate our proposed architecture, we implement our design in GloMoSim V2.03, a popular wireless network simulator. GloMoSim was developed as a modular library of components that contribute to an extensible, robust, and dynamic simulation of wireless networks. By isolating nodes' communication layers into independent modules, GloMoSim allows researchers to "plug and play" different protocols (i.e. protocols that they develop and implement) without concern for the inner workings of other architectural layers. The simulation environment allows us to apply our modules to the problem of event tracking. Specifically, by simulating event tracking in GloMoSim using our EMM and ECM modules, we are able to analyze the effects of architectural parameters on performance and understand how our architecture can be tuned to solve real world tracking problems.

4.1 Scenario

We modified the Transport and Application layers of GloMoSim to simulate a hypothetical sensor network environment consisting of nodes communicating over a 220 meter radius, which is typical for sensor nodes. The lower level protocols of our wireless network include Geographic Forwarding [16] over IP in the Network layer and 802.11B in the MAC layer. We simulated 32 byte packets sent in a 200 kbps wireless medium, a slightly larger bandwidth than the capabilities of today's sensor devices (50kbps-100kbps) to account for future improvements. At the physical layer, we use a Two-Ray Pathloss model with SNR-Bounded Noise at the receiving node. The model allows noise, attenuation, and subsequent loss on the wireless channel to be simulated, as opposed to perfect reception within a hypothetical radius.

Sensor nodes are equipped with sensors that poll their environment for specific events (e.g., acoustic sensors that monitor and can recognize certain acoustic signatures such as tank movement). In our experiments, a number of nodes are uniformly distributed in a 1,400 x 1,400 meter field. To test the ability of our architecture to track a moving target, we simulate an object moving across the field in a straight line. The moving object is tracked (presumably using acoustic or magnetic sensors) with a sensor polling period of 0.05 seconds, a granularity high enough to ensure up to date readings. Sensors register a target up to a sensing radius of approximately 100 meters.

For our tests we employ a simple application that computes an event's position through entity member reports to the entity leader. Entity members poll their sensors and send periodic updates to the corresponding entity leader notifying this leader of their current sensor reading and position. The leader computes the weighted average of the position of reporting entity members. The weighting is by sensor reading since higher readings presumably mean a closer target. This average value is sent back every 0.5 seconds to a "friendly-force" entity at a static location in the network. Upon receiving a report, this entity responds with a confirmation message sent back to the reporting entity. This feature allows us to test our architecture's ability to maintain end-to-end connectivity and forwarding between entities.

4.2 Simulation Results

The objective of our simulation is to understand the effect of algorithm parameters on tracking, as well as estimate costs such as the energy consumed. In all experiments an entity is spawned and migrates with the moving target. Entity uniqueness should be maintained for a run to be successful. Hence, we count the number of entities that form around the moving target during the course of a simulation to determine whether or not our architecture was successful in establishing and maintaining a single entity per event. In energy cost experiments, we compute the energy consumed during send and receive operations in accordance with transmit and receive currents of the MICA motes [23]. CPU energy consumed constitutes a constant overhead.

Each point in the graphs below represents the average of 10 runs to ensure a statistical significance at the 0.05 level. In the subsequent analysis when we claim a target is *trackable* at a specified speed we mean that for all 10 trials, a single entity was formed and tracked during each trial. The two key parameters of the algorithm, whose settings determine performance are the EMM leader heartbeat period and the awareness horizon. This leader heartbeat period defines how often the entity leader sends heartbeats to members (and followers). The awareness horizon, in our experiments, defines how many hops the heartbeats are propagated. Other algorithm parameters are automatically computed depending on the settings of the above two. Namely, we require that the failed leader timer (used to detect leader failure) be set to a value twice greater than the heartbeat period to ensure that no member takes over leadership while a current leader is still sending heartbeats. Similarly we require that the entity timeout period (used to free follower nodes) be approximately 1.5 times the failed leader period to ensure that no follower leaves the group before an entity member could properly take over leadership and begin sending heartbeats.

Below, we present and discuss those parameters that we feel are most influential to the problem addressed and the solution presented. We initially start with those parameters that are determined by the network or otherwise outside of the designer's control. We then analyze parameters that can be set by the designer. For these graphs we choose to display the heartbeat timer on the x-axis to analyze its affect on chosen metrics. We vary the range of heartbeat values from graph to graph to demonstrate trends in the timer and show what we feel is the most relevant and interesting information for understanding our architecture's performance.

4.2.1 Setting Node Density

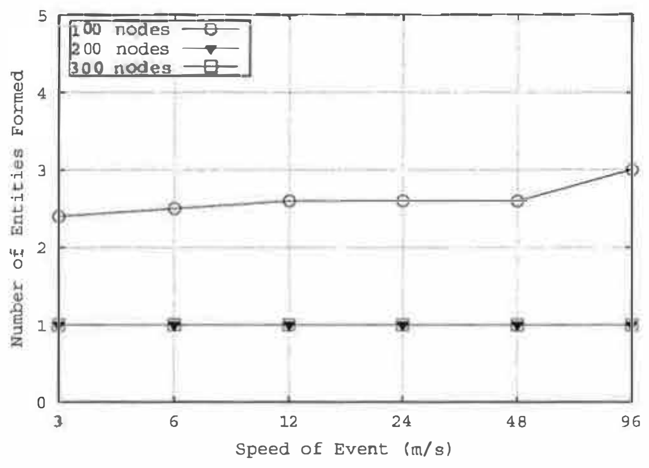


Figure 3. Effect of node density on number of groups formed (heartbeat period = 375 ms, awareness horizon = 1 hop)

It is often assumed in sensor network research that the node density is high enough to ensure that all nodes are within communication range of several other nodes at all times. We begin by understanding the effect of node density on the performance of our algorithm. In this experiment, we vary the number of nodes in the rectangular field, and observe the number of formed entities around the moving target. The experiment is repeated for different target speeds. A run is successful if only one entity is formed. In the experiments below, we choose the awareness horizon to be one hop, which as we show later, represents a worst case from the perspective of trackability at the target speeds considered.

Figure 3 demonstrates the results. We see that independent of event speed, our architecture is capable of maintaining the formed logical entity when the number of nodes is 200 or higher which corresponds to an average distance of about 140 meters between any two nodes. Thus, in the rest of this section we fixed the number of sensor nodes to 200, as a rough estimate of "sufficient" node density.

4.2.2 Effect of Leadership Handoff

Next, we explore the effect of the leadership handoff mechanism on performance. While our architecture includes a leadership handoff mechanism that explicitly notifies entity members when a new leader should be elected, we compare that mechanism to the case where the old leader simply dies in which case a timeout must elapse before the new leader election starts. Figures 4 and 5 compare the average number of entities formed around the target for different target speeds with and without the handoff mechanism respectively. For each target speed, we vary the leader heartbeat period. Larger periods mean that more time will elapse before leader failure is noticed. Observe that the curves in Figure 4 are generally lower than in Figure 5, indicating a larger fraction of successful runs. Remember that each point on each curve is the average of 10 runs. Points indicating a single formed entity mean that all 10 runs were successful. The handoff mechanism is more successful in tracking since it avoids the extra delay in leadership handoff making it less likely that the target will move to where it can be sensed outside of the awareness horizon, thus causing a spurious entity to emerge. Thus, in the rest of our analysis we only consider simulations where our handoff mechanism is present.

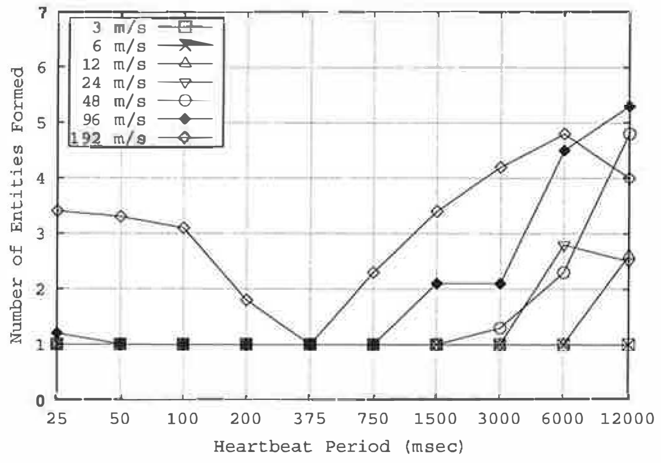


Figure 4. Groups formed with our explicit handoff mechanism (awareness horizon = 1 hop)

In addition, it is interesting how the choice of the heartbeat period strongly influences our architecture's ability to track an event. Slow periods will result in a slower transition resulting in the event migrating beyond the awareness horizon. Fast periods result in a congested channel which increases message loss and prevents nodes from hearing about an approaching event. In between these extremes, an optimal choice of the heartbeat period can be made. This choice will be investigated in more detail later in the section.

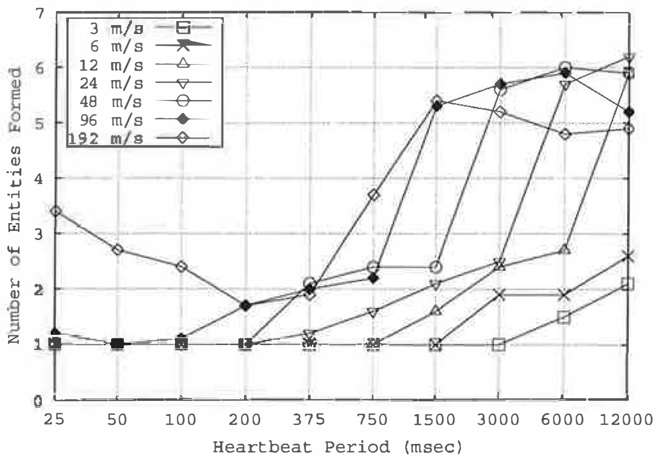


Figure 5. Groups formed without our explicit handoff mechanism (awareness horizon = 1 hop)

4.2.3 Limits on Heartbeat Period

Next, we analyze the overhead of the algorithm by exploring the point at which it saturates the underlying network. This saturation point depends on the setting of the heartbeat period and the awareness horizon. It is fairly obvious that decreasing the heartbeat period results in more frequent communication between the nodes and therefore the ability to track faster targets. This is based simply on the speed of response necessary for faster targets. However, the bandwidth limitations in the wireless medium place limits on our timer settings and constrain our architecture's ability to track migrating events. To determine the bandwidth needs of our algorithm, we start with a very small leader heartbeat period (that saturates the network), then increase it gradually. We plot the resulting connection delay, which is the time it takes to send a message from the moving entity to the friendly-force entity. As the heartbeat period increases to the point when the network is no longer saturated with heartbeat traffic, we observe a sharp decrease in the connection delay. Figure 6 shows this effect. The experiment is repeated for different awareness horizons, expressed in the number of hops that leader heartbeats are propagated to. It is seen that when the horizon is increased, the onset of overload occurs earlier as more messages are communicated.

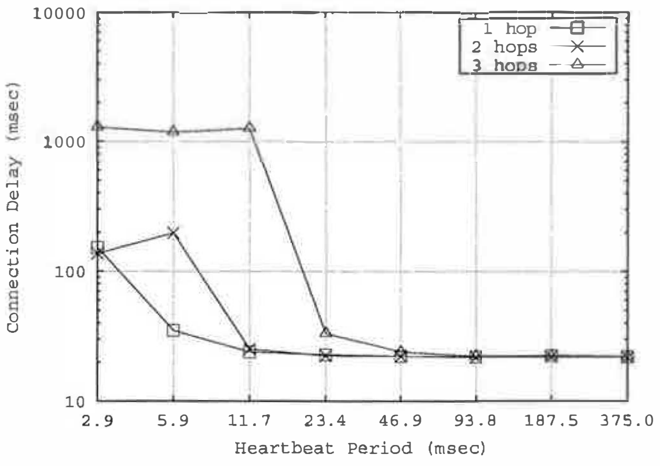


Figure 6. Effect of timer settings on message delay (speed = 12 m/s)

From Figure 6 we can see that the bandwidth of the wireless medium is fully saturated when the leader heartbeat period is reduced to approximately 2.9, 5.9, and 11.7 ms for an awareness horizon of = 1, 2, and 3 hops respectively. To conservatively avoid this saturation point and ensure enough bandwidth is left for alternate local traffic, we multiply these numbers by 5 (i.e., limit the worst case overhead of tracking to 20%). Hence, in the rest of the evaluation section, we consider only those leader heartbeat periods that are above 12.5, 25, and 50 ms for 1, 2, and 3 hops respectively. We next turn our attention to the selection of the leader heartbeat period and the awareness horizon, the two key parameters of the algorithm, subject to the above constraints.

4.2.4 Heartbeat Period and Awareness Horizon

Our group management protocol works by propagating leader heartbeats a specified number of hops from the leader, which determines the awareness horizon. Increasing the hop count allows us to track faster events since it extends the area in which nodes know about the oncoming event and requires a faster event to migrate faster than its corresponding logical entity. However as previously mentioned, the awareness horizon restricts the maximum leader heartbeat period, another important parameter of the algorithm. In the following experiment, we study the coupling between these parameters to analyze their affect on the maximum trackable event speed. Specifically, we vary the leader heartbeat period over a range above the pre-determined saturation points, and compute for each period the maximum event speed at which tracking always succeeds in maintaining a single entity for the moving target. A different curve is plotted for different awareness horizons.

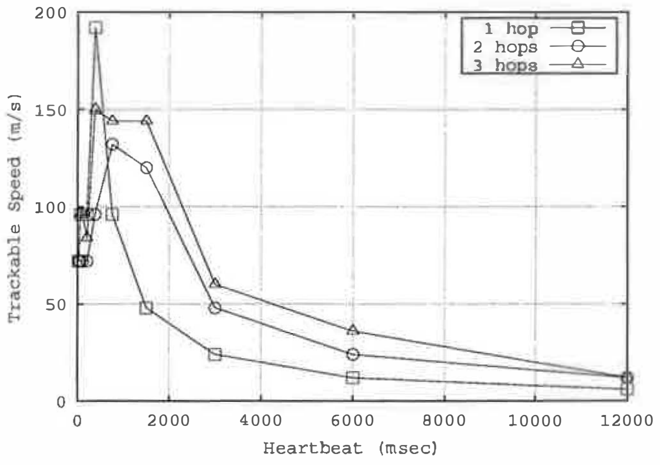


Figure 7. Trackable event speeds for varied heartbeat period:awareness horizon settings

Figure 7 shows the results of the above experiment. Following the curves from right to left, we can see that up to a threshold, reducing the leader heartbeat period results in the ability to track faster events. However, at this threshold, network collisions, congestion, and message loss become dominant. Hence, the maximum trackable speed deteriorates. This is apparent from the sudden drop in the maximum trackable speed seen on the far left side of the graph (about 100 ms). We also note that increasing the awareness horizon increases the trackable speed for slower heartbeats, but also congests the channel faster resulting in an inability to track faster targets. In the graph we notice that the single hop awareness horizon remains below the 2 and 3 hop settings up until about 400 ms, the point at which all but the 1 hop settings begin to break down.

A designer's decision to set the heartbeat period and the awareness horizon should be guided by the limitations on trackable speed shown in Figure 7. The trends seen in this figure illustrate the fundamental trade-offs involved in parameter selection in our algorithm. While the axes can be scaled depending on other platform settings such as node spacing, the figure provides interesting insights into the choice of protocol settings. For example, the figure shows that for speeds lower than 120 m/s (i.e., slightly more than half our communication radius per second) a designer has a choice of heartbeat period and awareness horizon settings that jointly allow a given speed to be tracked. The candidate settings are those that result from intersecting a horizontal line (drawn at the desired speed) with each of the three plots representing the different awareness horizons. Each intersection point gives the heartbeat period needed for the corresponding awareness horizon. In general, choosing a larger awareness horizon implies a more relaxed (i.e., larger) heartbeat period. For example, to track a target of speed 50 m/s, one can use a heartbeat period of 1600, 2900, and 4200 ms for a horizon of 1, 2, and 3 hops respectively. Interestingly, observe from Figure 6 that at those periods the network is fairly underloaded. When the target speed is higher than 120 m/s, however, larger values of awareness horizon fail, since they present a higher percentage of message loss and larger delays which allow spurious entities to be created. At those speeds a smaller awareness horizon is necessary.

If the designer has multiple choices, an important factor in choosing a particular combination of parameters is power consumption. The effect of parameter settings on power consumption is explored next.

4.2.5 Energy Consumption

Aside from their influence on the maximum trackable speed, the leader heartbeat period and the awareness horizon also significantly influence message related energy consumption. For different hardware this effect will vary. However, the fundamental trends remain consistent.

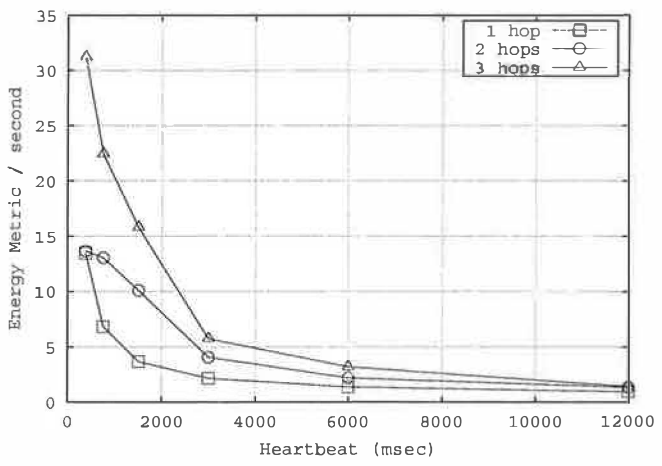


Figure 8. Energy consumption for varied heartbeat period:awareness horizon pairs

Figure 8 shows energy consumption for varying the leader heartbeat period and awareness horizons explored in Figure 7. The effect of the heartbeat period on energy consumption is significant. An interesting point to notice, however, is that the alternative (horizon, period) tuples that track a particular speed consume almost the same amount of energy. Following up on the example from the previous section, the three alternative parameter tuples that track a speed of 50 m/s, namely, (1 hop, 1600 ms), (2 hops, 2900 ms), and (3 hops, 4200 ms), consume roughly the same energy of 4 units. This is seen by finding the intersections of the vertical lines at 1600 ms, 2900 ms and 4200 ms with the energy curves for the corresponding horizon. It therefore appears that choosing a larger horizon does not have an advantage as long as the heartbeat period is appropriately chosen. This is intuitive. Increasing the awareness horizon allows using a larger heartbeat period. However, it also increases the number of messages sent, since each heartbeat is flooded in a larger radius. The net sum of messages exchanged, therefore, remains largely unaffected.

In view of the above, an argument can be made for smaller horizons, since they reduce the number of nodes inhibited from creating new entities, thus making the sensor network more responsive to the advent of new events into the environment. Nodes in a network with a large awareness horizon will attribute measurements of such new events to the entities they are already aware of; an effect that should be avoided.

The set of experiments presented above illustrate the ability of our architecture to track events at varied speeds in a sufficiently dense sensor network. A system designer can employ this architecture, choosing the leader heartbeat period and the awareness horizon in accordance with the expected speed of migrating events, available bandwidth of the wireless medium, and energy restrictions (required system lifetime) of the hardware being deployed. An implementation of our architecture on the MICA motes which is discussed in the following section.

5 Motes Implementation

We implemented our work on the MICA motes developed by X-Bow. Under our architecture we implemented a simple Geographic Forwarding [16] mechanism as well as a simple buffering MAC protocol that follows a send when ready strategy with no contention resolution. See [11] for more information on the MICA platform and TinyOS Operating System.

Our MICA test bed consists of 24 motes laid out in a 12 X 2 grid with motes placed one foot apart. Nodes communicate to neighboring nodes within a specified neighborhood grid size (NGS). The awareness horizon is set to 1 hop in accordance with the small size of the network. For our experiments we tested NGS sizes of 2 and 4.

Nodes are employed with light sensors capable of detecting a shadow cast by the tracked event. Our goal was to track a rectangular object 1 square grid in size, moving at a speed of 1 grid per 10 seconds (GrPS). Nodes are programmed with our tracking architecture and an application which performs target location estimation. Additionally, nodes are

Neighborhood	1/5 grid/s	1/10 grid/s	1/15 grind/s
2	5	4	1
4	3	1	1

Table 1. Entities formed over varied event speeds on the MICA testbed

pre-configured to report a detected event to a fixed location in the sensor network (node 0) every time the aggregate event location changes.

Based on the limited bandwidth, energy restrictions, expected target speed, and radio radius of the MICA motes, we experimented with and chose our heartbeat, failed leader, and entity timeout timers to be 2, 5, and 8 seconds respectively.

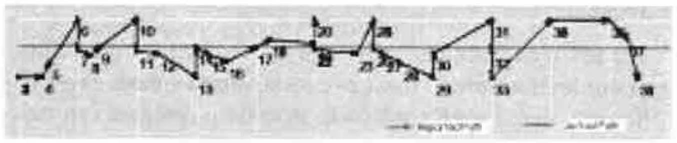


Figure 9. Reported location of tracked entity on MICA motes (Speed = 1/10 GrPS, NGS=4)

Initial experimental data show that for speeds of 1/10, and 1/15 GrPS and a NGS of 4 grids, our architecture as deployed was capable of correctly forming, maintaining, and reporting the location of a group around the event of interest. The reported path of the event for one such trial is shown in Figure 9. In this experiment, the tracked vehicle was traveling in a horizontal line from left to right as shown by the horizontal axis in the figure. The jagged quality of the reported path is a result of the limited number of motes detecting and reporting a position for the target at any specific point in time. It is possible that when the communication radius and the speed of events vary, multiple entities can be formed. Table 1 shows the number of entities formed over different speeds in the MICA testbed. The creation of spurious entities at faster speeds is due largely to the use of an unreliable MAC layer in the experiments. Message loss, as mentioned in the paper, allows spurious entities to be created. The noisy nature of the measured track is due to the fact that the light sensors do not provide a measure of distance from the target. Hence, the best one can do is simply average the positions of all sensors detecting the event. In contrast, other sensors (such as magnetic senors) can provide a better estimate of how far a measured target is. Triangulation will therefore result in a much more accurately estimated position.

Observe that the purpose of our experimental measurements is to qualitatively illustrate the success of our scheme in practice. It is not our purpose to compare experimental measurements to simulation. We have purposely decided to use a standard wireless MAC layer in the simulator instead of the simplified unreliable custom MAC layer implemented on the actual motes. Consequently, different performance results are expected. We believe that the implementation of the motes MAC layer will mature significantly in the future, making it less interesting to seek quantitative performance statements on the current testbed at this time.

6 Conclusions and Future Work

In this work we provide a transport layer solution to entity maintenance and connectivity in sensor networks. Our proposed middleware service provides a novel way for programmers to relate to events in the environment without concern for topological and communication layer details irrelevant to their application. We establish entities in a one-to-one relationship with events to ensure correct and well defined behavior. Entities form and register with interested parties to allow unique identification and communication without regard for an event's location.

The analysis of our architecture demonstrates the effect of both uncontrollable (environment specific) and controllable (architecture specific) parameters on entity formation and maintenance. In accordance with an ideal sensor network, we require that at any time at least one node is capable of sensing the event being tracked. Under this assumption we demonstrate our architecture's capabilities and limitations in tracking events of varied speeds as a function of the pre-specified heartbeat and awareness horizon parameters. For a large field of relatively dense nodes, we show in simulation that our architecture is capable of tracking events that travel over half of the communication radius of a node per second (see Figure 7). Under optimal conditions in our simulation study, our architecture was able to track objects moving at about 88% of the communication radius. We additionally discuss the required settings for tracking events of varied speeds and the tradeoff of increasing the trackable speed and thereby increasing the amount of energy consumed.

In addition to simulation analysis, we provide an implementation of our architecture on the MICA test bed in our lab. In the presence of true fading and message loss we demonstrate the feasibility of our work for a simple application. Implementation results show the importance of a reliable MAC layer. Unlike the simulation, which used 802.11 to access the medium, the MAC layer in the implemented prototype used unreliable transmission. Consequently, the maximum tracked speed was significantly lower. The authors are currently implementing a reliable MAC layer for the motes platform. While our architecture provides the required mechanisms to create, maintain, and communicate with abstract entities in an environment, we have only begun to explore the possibilities of such sensor network related services. The applications and opportunities for sensor networks remain vast and mostly unexplored. This paper is a step towards a comprehensive coverage of research issues motivated by tracking problems in sensor networks.

Acknowledgements

This work was supported in part by DARPA grant F33615-01-C-1905, MURI grant N00014-01-1-0576, and NSF grant CCR-0208769

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan and J. Lilley, "The design and implementation of an intentional naming system," *Proceedings of 17th ACM SOSP*, Kiawah Island, SC, Dec. 1999.

[2] R. Barr, J. Bicket, D. Dantas, B. Du, T. Kim, B. Zhou and E. Sirer, "On the Need for System-Level Support for Ad hoc and Sensor Networks," *Operating Systems Review, ACM*, 36(2):1-5, April 2002

[3] S. Basagni, I. Chlamtac, V. Syrotiuk and B. A. Woodward, "A Distance Routing Effect Algorithm for Mobility (DREAM)," *ACM/IEEE Mobicom 1998*, pp. 76-84, Dallas, TX, October 1998.

[4] N. Bulusu, J. Heidemann and D. Estrin, "GPS-less Low Cost Outdoor Localization For Very Small Devices," *IEEE Personal Communications, Special Issue on "Smart Spaces and Environments"*, 7(5):28-34, October 2000.

[5] A. Cerpa and D. Estrin, "ASCENT: Adaptive Self-Configuring Sensor Networks Topologies," *IEEE Infocom 2002*, New York, NY., USA, June 23-23, 2002.

[6] B. Chen, K. Jamieson, H. Balakrishnan and R. Morris, "SPAN: An energy efficient coordination algorithm for topology maintenance in ad hoc wireless networks," *ACM/IEEE Mobicom 2001*, pp. 85-96, Rome, Italy, July 2001.

[7] D. Coore, R Nagpal and R Weiss, "Paradigms for Structure in an Amorphous Computer," AI Memo 1614, MIT, October, 1997.

[8] D. Estrin, R. Govindan, J. Heidemann and S. Kumar, "Next Century Challenges: Scalable Coordination in Sensor Networks," *ACM/IEEE Mobicom 1999*, pp. 263-270, Seattle, Washington, August 1999.

[9] Z. J. Haas and M. Pearlman and Prince Samar, "The Zone Routing Protocol (ZRP) for Ad-Hoc Networks," Internet Draft, July 2002.

[10] W. Heinzelman, A. Chandrakasan and H. Balakrishnan, "Energy-efficient communication protocol for wireless sensor networks," *Proc. Hawaii Intl. Conf. System Sciences*, pp. 3005-3014, Hawaii, 2000.

[11] J. Hill, R. Szewczyk, A. Wood, S. Hollar, D. Culler and K. Pister, "System Architecture Directions for Networked Sensors," *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, M.A., November 2000.

[12] M. Horton, D. Culler, K. Pister, J. Hill, R. Szewczyk and A. Woo. "Mica: The commericialization of microsensor motes," *Sensors Online*, 19(4), April 2002. http://www.sensormag.com/articles/0402/index.htm.

[13] C. Intanagonwiwat, R. Govindan and D. Estrin, "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks," *ACM/IEEE Mobicom 2000*, Boston, MA, August 2000.

[14] D. Johnson and D. Maltz "Dynamic Source Routing (DSR) in Ad Hoc Wireless Networks," *Mobile Computing* pp. 153-181, 1996.

[15] F. Kamoun and L. Kleinrock, "Hierarchical Routing for Large Networks: Performance Evaluation and Optimization," *Computer Networks 1977(1)*, pp. 155-174.

[16] B. Karp and H. T. Kung, "Greedy perimeter stateless routing (GPSR) for wireless networks," *ACM/IEEE Mobicom 2000*, pp. 243–254, Boston, MA, August 2000.

[17] Young-Bae Ko and Nitin H. Vaidya, "Location-Aided Routing(LAR) in Mobile Ad Hoc Networks," *ACM/IEEE Mobicom 1998*, Dallas, TX, October 1998.

[18] P. Levis and D. Culler "Mate? - a Virtual Machine for Tiny Networked Sensors," *ASPLOS*, Dec. 2002

[19] J. Li, J. Jannotti, D. S. J. De Couto, D. R. Karger, and R. Morris, "A Scalable Location Service for Geographic Ad hoc Routing," *ACM/IEEE Mobicom 2000*, pp. 120-130, Boston, MA, August 2000.

[20] C. R. Lin and M. Gerla, "Adaptive Clustering for Mobile Wireless Networks," *IEEE Journal on Selected Areas in Communication 1997*, 15(7).

[21] A. Mainwaring, J. Polastre, R. Szewczyk and D. Culler "Sensor Networks for Habitat Monitoring," *2002 ACM International Workshop on Wireless Sensor Networks and Applications*, Sept. 2002.

[22] A. B. McDonald and T Znati, "A Mobility Based Framework for Adaptive Clustering in Wireless Ad-Hoc Networks," *IEEE Journal On Selected Areas in Communication*, 17(8), August 1999.

[23] MICA: Wireless Measurement System. URL: http://www.xbow.com/Products/Product_pdf_files/, Crossbow Technology Inc., San Jose CA.

[24] R. Nagpal and D. Coore, "An Algorithm for Group Formation in an Amorphous Computer," *Proc. 10th International Conference on Parallel and Distributed Computing Systems (PDCS'98)*, Nevada, Oct 1998.

[25] V. Park and M.S. Corson, "A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks," *IEEE Infocom 1997*, April 1997.

[26] C. Perkins, Ed., "IP Mobility Support," RFC 2002, IETF, Oct. 1996.

[27] C. Perkins and P. Bhagwat, "Highly dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers," *ACM Sigcomm 1994*, pp. 212-225, London, UK, Sept. 1994.

[28] C. Perkins and D. Johnson, "Route Optimization in Mobile IP," Internet Draft, September 2001.

[29] C. Perkins and E. Royer, "Ad-hoc On Demand Distance Vector Routing," *2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'99)*, New Orleans, Louisiana, February 1999.

[30] Radihika Nagpal, "Organizing a Global Coordinate System from Local Information on an Amorphous Computer," AI Memo 1666, MIT, August 1999.

[31] R. Ramanathan and M. Steenstrup, "Hierarchically-organized, multiphop mobile wireless networks for quality of service support," *Mobile Networks and Applications 1998.*

[32] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker, "A Scalable Contentaddressable network," *ACM Sigcomm 2001*, San Diego, CA, Aug. 2001.

[33] A. Rosenstein, J. Li and S. Y. Tong "The Multicasting Archie Server Hierarchy," Project Home-Page. URL: http://www.cs.ucla.edu/ adam/mash.html.

[34] A.C. Snoeren and H. Balakrishnan, "An end-to-end approach to host mobility," *ACM/IEEE Mobicom 2000*, Boston, MA, August 2000.

[35] Ion Stoica, Daniel Adkins, Shelley Zhaung, Scott Shenker, and Sonesh Surana, "Internet Indirection Infrastructure," *ACM Sigcomm 2002*, pp. 73-86, Pittsburgh, PA, August 2002.

[36] I. Stoica, R. Morris, D Liben-Nowell, D. Karger, M.Kaashoek, F. Debek and H. Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications," *To Appear in ACM/IEEE Transactions on Networking*

[37] L. Subramanian and R. H. Katz, "An Architecture for Building Self-Configurable Systems," *ACM/IEEE Workshop on Mobile Ad Hoc Networking and Computing*, August 2000.

[38] D. G. Thaler and C. V. Ravishankar, "Distributed top-down hierarchy construction," *IEEE Infocom 1998*, San Francisco, CA, March 1998.

[39] P. F. Tsuchiya, "The Landmark Hierarchy: A New Hierarchy for Routing in Very Large Networks," *ACM Sigcomm 1988*, Stanford, CA, August 1988.

[40] N.H. Vaidya P. Krishna, M. Chatterjee and D.K. Pradhan, "A Cluster-Based Approach for Routing in Dynamic Networks," *ACM Computer Communications Review*, 27(2), April 1997.

[41] A. Woo and D. Culler, "A Transmission Control Scheme for Media Access in Sensor Networks," *ACM/IEEE Mobicom 2001*, Rome, Italy, July 2001.

[42] A. Wood and J. Stankovic. "Denial of Service in sensor networks" *IEEE Computer*, 35(10), October 2002.

[43] Y. Xu, J. Heidemann and D. Estrin, "Geographic-informed Energy Conservation for Ad Hoc Routing," *ACM/IEEE Mobicom 2001*, Rome, Italy, July 2001.

[44] F. Ye, H. Luo, J. Cheng, S. Lu and L. Zhang, "A Two-tier Data Dissemination Model for Large-scale Wireless Sensor Networks," *ACM/IEEE Mobicom 2002*, Atlanta, Georgia, Sep. 2002.

[45] W. Ye, J. Heidemann, D. Estrin, "An Energy-Efficient MAC Protocol for Wireless Sensor Networks," *IEEE Infocom 2002)*, New York, USA, June 2002.

Operating System Modifications for Task-Based Speed and Voltage Scheduling

Jacob R. Lorch
Microsoft Research
1 Microsoft Way
Redmond, WA 98052
lorch@microsoft.com

Alan Jay Smith
University of California, Berkeley
EECS Department, Computer Science Division
Berkeley, CA 94720-1776
smith@eecs.berkeley.edu

Abstract

This paper describes RightSpeed, a task-based speed and voltage scheduler for Windows 2000. It takes advantage of the ability of certain processors, such as those from Transmeta and AMD, to dynamically change speed and voltage and thus to save energy while running more slowly. RightSpeed uses PACE, an algorithm that computes the most energy efficient way to meet task deadlines with high probability. Since most applications do not provide enough data about tasks, such as task deadlines, for PACE to work, RightSpeed uses simple and efficient heuristics to automatically detect task characteristics for such applications. RightSpeed has only 1.2% background overhead and its operations take only a few microseconds each. It even performs PACE calculation, which is quite complicated, in only 4.4 μs on average due to our extensive optimizations. RightSpeed is effective at meeting performance targets set by applications to within 1.5%. Although the PACE calculator does not save energy for the current generation of processors due to their limited range of worthwhile speed and voltage settings, we expect future processors to have greater such ranges, enabling PACE to reduce CPU energy consumption by 6.1–8.7% relative to the best standard algorithm. Furthermore, with PACE, giving a processor the ability to run at additional, higher speeds and voltages *reduces* overall energy consumption.

1 Introduction

Reducing energy consumption is important in portable computers due to their limited battery capacity. Furthermore, rising concerns about energy prices and aggregate energy dissipation in server farms make energy management important for other computers as well. An energy-saving technology that has recently begun appearing in modern portable computers is dynamic voltage scaling (DVS), the ability to change processor voltage without rebooting. This enables reduced energy consumption, as lower voltages mean lower energy consumption. Lower voltages, however, necessitate lower CPU speeds, presenting an interesting operating system issue: how to ensure that performance remains reasonable while sometimes lowering speed to save energy.

Traditionally, systems use *interval-based* strategies. Such strategies divide time into intervals of fixed length and set the speed for each interval based on recent CPU utilization. However, recent CPU utilization is only a rough indicator of the required speed. An interval-based strategy cannot distinguish an urgent task that must run at full speed to meet a tight deadline from a less important task with several milliseconds to complete and little work to do.

A better solution, as suggested by authors such as Pering et al. [18] and Hong et al. [6], is to use *task-based scheduling*. Such scheduling considers the computer's work to consist of tasks with certain CPU requirements and deadlines. It then runs the CPU fast enough to meet those deadlines with reasonable probability. Recently, some researchers have even built such task-based schedulers [19, 4, 3]. In this paper, we describe how we built RightSpeed, a task-based scheduler with several improvements over these existing schedulers.

The key differentiating feature of RightSpeed is its *PACE calculator*, a component that determines the most energy efficient schedule for meeting each task's performance requirements. PACE stands for Processor Acceleration for Conserving Energy, since the optimal way to schedule a task is to start out slowly, increasing speed only as necessary to complete the task on time. In [11], we showed that computing such a schedule requires estimating the probability distribution of the task's CPU requirement, and gave a method called PACE that uses such a distribution to compute such a schedule. For this paper, we extended this method substantially to deal with issues that arise in real systems: limited available speed/voltage settings, nonlinear relationship between speed squared and energy, limited timer granularity, and I/O wait time.

RightSpeed also differs from other schedulers in the heuristic it uses for *automatic task detection.* A task-based scheduler can provide an interface letting applications specify information about their tasks. However, many application writers will not use it, so a task-based scheduler should also have an automatic task detector to let it infer task information from such applications. The schedulers Flautner et al. describe in [4] and [3] have such detectors, but they require a great deal of complex, high-overhead, and Linux-specific system interposition. In [12], we suggested a method for automatic task detection with a more efficient heuristic, but did not demonstrate an implementation. RightSpeed demonstrates an implementation of our heuristic.

Our scheduler also differs from existing schedulers by running on Windows 2000 rather than Linux. This is important because most portable computers sold today run Windows 2000 or its successor Windows XP. Our work demonstrates that task-based scheduling can be done even on a closed-source commodity operating system.

The goal of this paper is to demonstrate that a task-based scheduler with a PACE calculator and an automatic task detector can be implemented on a real machine running Windows 2000. This involves overcoming the challenges of real hardware and software issues, and demonstrating that the resulting scheduler places little overhead on the system.

The structure of this paper is as follows. Section 2 gives background and related work on DVS algorithms. Section 3 describes the characteristics of the processors to which we ported RightSpeed, and evaluates the potential effectiveness of DVS techniques on these processors. Section 4 discusses the design of our task-based scheduler, and Section 5 describes our implementation of it. Section 6 gives results of benchmarks showing the impact of our modifications on performance and energy consumption. Section 7 discusses avenues for future work. Finally, Section 8 concludes.

2 Background and Related Work

2.1 Dynamic voltage scaling

In CMOS circuits, the dominant component of power consumption is proportional to $V^2 f$, where V is voltage and f is frequency. Energy is power times time, and the time to run a certain number of cycles is inversely proportional to frequency, so energy per cycle is proportional to V^2 [22, p. 235]. At a given voltage, the maximum frequency at which the CPU can run safely decreases with decreasing voltage. Thus, the system can reduce processor energy consumption by reducing CPU voltage, but this necessitates running at a slower speed.

However, it is important to not noticeably increase system response time, for two reasons. First, other components, such as the disk drive and backlight, use power. Noticeably increasing response time may cause these components to remain in high-power modes longer than they otherwise would, which can more than offset processor energy savings. Second, the user will object to unduly extended response times.

2.2 Interval-based DVS algorithms

The first researchers to discuss operating system techniques for DVS were Weiser et al. [21] and Chan et al. [2]. They suggested an interval-based approach, meaning that the system divides time into fixed-length intervals and schedules the speed for each interval based on the CPU utilizations of past intervals.

Interval-based strategies are used today in real systems capable of dynamic voltage scaling, such as Transmeta's LongRun™ [7]. However, such strategies have problems, as Pering et al. [17], and later Grunwald et al. [5], pointed out. The CPU utilization by itself does not provide enough information about system timing requirements to ensure meeting a reasonable number of deadlines while saving energy.

2.3 Task-based voltage schedulers

Recently, researchers have started building task-based schedulers, i.e., schedulers that consider the work of the system to consist of tasks with certain deadlines. The goal of a task-based scheduler is to use speeds just high enough to meet these deadlines with reasonable probability.

Yao et al. [23] described how to compute an optimal schedule when task CPU requirements and deadlines are known. Hong et al. [6] later showed how to compute such schedules more quickly using various heuristics. However, systems do not generally have definite knowledge of task CPU requirements, so these approaches are unrealistic.

Flautner et al. [4] built a task-based voltage scheduler for Linux. This scheduler requires no modification of applications—it infers all information about the system's tasks via heuristics. It infers that an interactive task begins when a user interface event arrives, and uses a complex work-tracking heuristic to decide when such a task completes. It infers that a periodic task begins when a periodic event occurs; it considers an event periodic if the lengths of intervals between the last n events have a small variance. To determine the speed for a task, it essentially computes the average of the speeds that would have completed past similar tasks on time. In later work [3], they refined their period detector (but not their task completion detector) to use a simpler heuristic and they extended their interactive performance-setting algorithm with two other policy layers: one for application-specific policies and one for a per-task interval-based policy.

Pillai et al. [19] built a task-based scheduler for real-time embedded systems that runs on Linux. This scheduler assumes complete knowledge of the deadlines and worst-case CPU requirements of all tasks in the system, and assumes these tasks are periodic. The scheduler uses different algorithms, some of which make provisions for tasks completing before their deadlines. One such algorithm slows down the CPU when a task creates slack in the schedule by completing early. Another algorithm anticipates that tasks will likely complete early and therefore starts tasks as slowly as possible and only uses higher speeds when these become necessary to guarantee on-time completion.

2.4 PACE

One premise of task-based scheduling is that DVS can exploit deadlines to save energy without significantly reducing performance. This is possible since a task's completion time is irrelevant as long as it completes before the deadline. Thus, in evaluating the performance of a DVS algorithm, we can consider all tasks that complete by the deadline to have the same effective performance.

A DVS algorithm essentially chooses a schedule describing how speed will vary with time. In [11], we showed that two schedules that have the same average pre-deadline speed and identical post-deadline parts will give the same effective performance no matter how much work a task requires. This means that one can get the same performance as any existing DVS algorithm by using different, yet performance equivalent, speed schedules; these new schedules may even consume less energy.

We then described an algorithm, PACE, for choosing a speed schedule that minimizes expected energy consumption for a given performance constraint. The PACE algorithm assumes some knowledge of task CPU requirement distribution; we showed how to dynamically and effectively estimate this distribution. One limitation is that PACE assumes the processor speed and voltage are continuously variable and that energy is a linear function of speed squared; in this work, we extend PACE to real DVS systems without these properties.

PACE requires the ability to detect when tasks begin and end. In [12], we showed that there is a simple heuristic for inferring task completion that is nearly as effective as Flautner et al.'s scheduler [4] and requires substantially less operating system modification. Our approach considers a task complete when either all threads in the system are blocked and no I/O is ongoing, or when a new user interface event is delivered to the same application.

In [12], we pointed out that user interface events belonging to different types, categories, and applications differ significantly from each other. This difference is large enough that PACE benefits, rather than worsens, by inferring the probability distribution of a task from a sample of only those recent past tasks that have nearly identical characteristics. Therefore, in RightSpeed, we keep separate samples for tasks triggered by user interface events of different types, categories, and applications.

3 Platforms

In this section, we examine the characteristics of Transmeta and AMD processors to which we ported RightSpeed. As we do so, we will discuss how these characteristics influence how we should use PACE on these processors.

First, we introduce some definitions. A *setting* is a speed and voltage combination at which a processor can properly operate. The *efficiency* of a setting is the amount by which power consumption is reduced by using this setting instead of emulating its speed using the best possible combination of all other settings. For example, suppose there are three settings: 300 MHz consuming 2 W, 500 MHz consuming 3.6 W, and 700 MHz consuming 6 W. We can emulate 500 MHz by running half the time at 300 MHz and half the time at 700 MHz. This consumes 4 W, while the 500 MHz setting consumes only 3.6 W, so the 500 MHz setting has efficiency 10%. We can emulate 300 MHz by running 60% of the time at 500 MHz and turning the CPU off 40% of the time; this emulation has average power consumption 2.16 W., so the 300 MHz setting has efficiency 7.4%. If a setting has efficiency of 0% or less, it is not *worthwhile*, i.e., one should never use it since one can get lower power consumption at the same speed using other settings.

For PACE to be effective, a processor must have at least three worthwhile speed/voltage settings. Furthermore, the more settings, and the higher their efficiency, the more effective PACE will be. This is because PACE works by choosing among speed schedules with identical performance to find the one with least expected energy consumption. If there is little choice in such speed schedules, and/or if there is little difference between choosing one setting versus emulating that setting's speed with other settings, there will likely be little benefit to choosing among them.

Speed	Voltage	Power	Energy/ cycle	Efficiency
297.3 MHz	1.2 V	1.349 W	4.537 nJ	0.5%
396.6 MHz	1.225 V	1.809 W	4.561 nJ	11.0%
497.8 MHz	1.35 V	2.714 W	5.461 nJ	11.8%
598.5 MHz	1.55 V	4.348 W	7.265 nJ	0.4%
631.1 MHz	1.6 V	4.915 W	7.787 nJ	N/A

Table 1: Characteristics of the Transmeta processor at various settings

Speed	Voltage	Power (est.)	Energy/ cyc (est.)	Efficiency (est.)
500 MHz	1.25 V	10.6 W	21.3 nJ	7.6%
600 MHz	1.3 V	13.8 W	23.0 nJ	1.4%
700 MHz	1.35 V	17.4 W	24.8 nJ	-0.9%
800 MHz	1.4 V	21.3 W	26.7 nJ	-3.6%
900 MHz	1.4 V	24.0 W	26.7 nJ	N/A

Table 2: Characteristics of the AMD processor at various settings, with power and energy values approximated

3.1 Transmeta system

Our Transmeta system contains a TM5400-633 Crusoe^{TM} processor and 128 MB of memory (64 MB of SDRAM and 64 MB of DDRAM). 16 MB of this memory is reserved for the Code-Morphing Software, whose primary function is to dynamically translate x86 code to the underlying machine language of the VLIW chip. This code also implements LongRun^{TM}, the DVS policy Transmeta chips use. Transmeta told us how to override LongRun^{TM} policies and change the speed ourselves.

The processor can run at 300–633 MHz and 1.2–1.6 V. Table 1 gives the available speeds and voltages, as well as the power the CPU consumes at each level. We measured power consumption by running a tight loop of additions while using hardware monitoring equipment Transmeta provided.

We see that the 300 MHz and 600 MHz settings have very low efficiencies, and are therefore barely worthwhile. With only three reasonably worthwhile settings, we do not expect PACE to be very effective on this machine.

Incidentally, we note that the formula $1.179 \cdot 10^{-9} \cdot s^{3.41} + 3.681$, where s is speed, gives a very close approximation to the energy consumption in nJ/cycle for all but the 300 MHz setting. The power of 3.41 differs substantially from the power 2 predicted by simple scaling models, e.g., in [21].

3.2 AMD system

Our AMD system contains a pre-production version of the 900 MHz Mobile Athlon 4 processor, based on the Palomino core, as well as 128 MB of memory. We were given documentation about PowerNow!^{TM}, the interface the chip uses for dynamically changing speed and voltage.

The chip indicates it is capable of five settings, shown in Table 2. We were unable to directly determine the power consumption of each setting since we lacked the necessary measurement equipment, so we estimate it using $P \propto V^2 f$. We assume a power consumption of

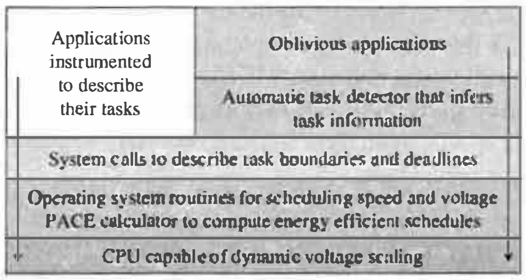

Applications instrumented to describe their tasks
Oblivious applications
Automatic task detector that infers task information
System calls to describe task boundaries and deadlines
Operating system routines for scheduling speed and voltage
PACE calculator to compute energy efficient schedules
CPU capable of dynamic voltage scaling

Figure 1: Overview of RightSpeed

24 W at the maximum speed, as specified in the AMD data sheet [1].

We see that the 700 MHz and 800 MHz settings have negative efficiency, so they are not worthwhile. (It is not surprising that the 800 MHz setting is not worthwhile, since it has the same voltage as the 900 MHz setting, and thus the same energy consumption, but it runs more slowly.) Furthermore, the 600 MHz setting has rather low efficiency. With only three worthwhile settings, one of which is only barely worthwhile, we expect PACE to be largely ineffective.

We suspect that some settings have poor efficiency because AMD made overly conservative choices of maximum stable speed for certain voltages. One reason for this is that their current design requires processor speeds and voltages to attain only a certain set of values. More flexibility in either dimension would let them choose settings closer to the curve of maximum ideal efficiency.

4 Design

4.1 Overview

Figure 1 gives an overview of the RightSpeed design. Applications convey information about their tasks to the operating system using system calls. This information includes when tasks begin and end and what performance targets the application wants for those tasks. Some applications are oblivious to the existence of these system calls, so an automatic task detector infers task information about them and generates task specification system calls on their behalf. The system uses information about ongoing tasks to determine what speed to use at various times, and implements this schedule using timers and special processor instructions that change speed and voltage. The system uses a PACE calculator to compute the most energy efficient schedules that have the performance requested.

In addition to the above functionality, we had three overall goals for RightSpeed. First, we wanted it to be efficient, creating low overhead on the system both when running in the background and when actively invoked. Second, we wanted it to be stable, relying only on documented system interfaces so that it would run even when the operating system was upgraded. Third, we wanted it to be easily portable to different processors despite such processors having different commands dealing with speed and voltage settings.

4.2 Task specification interface

A key piece of information an application must specify about a task is its *type*. An application may define types any way it chooses; there are two reasons applications will want to classify different tasks into different types. First, it may want to specify different performance targets for different types of tasks. For example, an MPEG player may require a faster speed for processing its I-frames than its smaller P-frames. As another example, it may want a short and hard deadline for its frame playback tasks but a longer and soft deadline for its user interface tasks. Second, tasks of different types may have different CPU requirement distributions, so it is helpful to direct PACE to only consider tasks of the same type when estimating the probability distribution of a task's CPU requirement.

RightSpeed uses this notion of task type to simplify its communication with applications. When an application begins a task, it need only tell RightSpeed the type of that task. RightSpeed can figure out all other information about the task, such as its performance requirements, from that type. RightSpeed can give the application a unique identifier to identify this task, so the application can specify when the task completes by merely passing RightSpeed that identifier. RightSpeed can then determine how many CPU cycles that task used and use this datum to compute a new optimal PACE schedule for the next task of that type.

An application specifies performance targets for task types via a separate part of the task specification interface. An application need only specify this data once, when it is installed. Because task type data is persistent, i.e., it is retained even when the application terminates and even when the system shuts down, a logical abstraction to use for this data is a file. Thus, applications create files containing data for their task types.

An application may specify a performance target in two ways. First, it may specify a number of CPU cycles to be completed by a certain deadline. Second, it may specify a deadline and a particular DVS algorithm, such as Transmeta's LongRun^{TM}, and dictate that performance be the same as would be achieved via that algorithm.

4.3 Automatic task detector

Since RightSpeed has not been released, no application currently exists that explicitly communicates its task information to RightSpeed. Furthermore, even when it is released, we expect few application writers will be both willing and able to communicate such information. Therefore, for RightSpeed to be useful, we require an automatic task detector to infer task information from such applications and to call the task specification interface on their behalf.

Our approach focuses on the tasks the user cares about most: those triggered by user interface events. User interface studies have shown that response times under 50–100 ms do not affect user think time [20]; we thus consider 50 ms the soft deadline for handling a user interface event. An exception is mouse movements, whose tracking may require response times of only 25–50 ms [13]; we thus consider 25 ms the soft deadline for handling them.

We consider a task to begin when an application receives a user interface event. We classify tasks into types, and deduce the task type from the event characteristics, i.e., whether it is a keystroke, mouse movement, or mouse click; which key or mouse button was pressed or released; and to what application the event was delivered. As shown in [12], separating tasks into types this way makes estimation of task work distribution more accurate, and enables us to set different policies for, for instance, keystrokes and mouse clicks.

As suggested in [12], we use the minimum speed available as the pre-deadline speed for mouse movement events. Such events require little processing, so this is sufficient to meet practically all task deadlines. We use a default pre-deadline speed of $0.7M$ for keystroke events and $0.85M$ for mouse click events, where M is the maximum speed available on the machine. A better approach might be to compute a variable pre-deadline speed based on the distribution observed and the likelihood of missing deadlines at various pre-deadline speeds, as suggested in [12]. Unfortunately, this requires accurate estimation of the tails of nonstationary distributions, and we do not yet know how to do this; this is future work.

We also need a heuristic to determine when such an inferred task is complete, since it is difficult to determine what CPU activity belongs to a given task. We use the heuristic from [12] described in Section 2.4: we consider a task complete when either (a) all threads in the system above the idle priority level are blocked and no I/O is ongoing, or (b) another user interface event is delivered to the same application. An advantageous side effect of this is that time spent by unrelated threads is considered part of the task. Thus, the speed schedule chosen will automatically account for the work performed by other threads during the task. Without this accounting, the presence of such unrelated activity could interfere with RightSpeed meeting its target deadlines.

4.4 PACE calculator

Computing the optimal speed schedule satisfying certain performance constraints requires knowledge of task CPU use distribution, which typically an application lacks. RightSpeed keeps track of how long tasks of each type have taken, and uses this information to compute such an optimal speed schedule with PACE.

In [11], we described how to compute an optimal schedule assuming a linear relationship between energy and speed squared. Since the processors on which RightSpeed runs do not satisfy this property, we developed a more general formula that does not rely on it. We discovered that the optimal speed schedule satisfies $s^2 \mathsf{E}'(s) F^c(w) = K$, where s is the speed to run after completing w cycles of a task, $F^c(w)$ is the probability the task takes more than w cycles, $\mathsf{E}(s)$ is the energy consumption at speed s, and K is a constant chosen to satisfy the performance constraint. For more details about this formula and a proof that it works, see [9, pp. 83–99].

In [11], we assumed that the CPU had arbitrarily variable speed settings that could be changed at arbitrary times. Our real systems have only a limited number of speed settings, and Windows 2000 only allows us to change speed at certain fixed times, once per millisecond. Thus, for RightSpeed we need an algorithm that takes these realities into account yet still computes a near-optimal schedule. Our algorithm uses the following four steps.

1. Create an idealized schedule using the formula above. Apply the granularization techniques of [11] to get a schedule consisting of consecutive *phases*, each having a constant speed.
2. For each phase, round its speed to the closest speed that is available on the CPU and worthwhile.
3. Round the length of each phase to an integer multiple of the scheduling granularity.
4. As the rounding may have altered the schedule's performance characteristics, i.e., changed the pre-deadline speed, adjust the time spent at each speed by multiples of the scheduling granularity to make performance close to, but no less than, requested performance.

As an optimization, we precompute a set of parameterized speed schedules when RightSpeed is installed, based solely on the CPU characteristics. Thus, determining a speed schedule involves only a binary search through the schedules to find the lowest-energy one that nevertheless satisfies the constraint. With this optimization, the algorithm takes time $O(n)$ where n is the number of worthwhile speed settings. For details of this and other optimizations, see [9, pp. 224–226] and the code at the website associated with this paper.

4.5 Dealing with I/O

I/O time, unlike CPU time, is unaffected by changes in CPU speed. The model from which PACE arises accounts only for task CPU time, so PACE does not give optimal results when I/O can occur. Essentially, the occurrence of I/O will delay the completion of a task, possibly causing it to miss its deadline.

We deal with this in the following way. Since the problem is to complete the CPU work *and* the I/O by the deadline, we must complete the CPU work within a period equal to the deadline minus the I/O time. If we knew I/O time in advance, PACE could compute the optimal schedule merely by substituting the deadline minus I/O time for the deadline. Since we do not know I/O time in advance, we initially assume it is 0. If I/O occurs later, we determine how long it took and accelerate the schedule to make up for the lost time.

Theoretically, accelerating the schedule properly requires performing a new complex calculation using the PACE formula. However, we can use a shortcut: we multiply all speeds in the schedule by a constant factor, where we choose that factor such that after rounding all resulting speeds to the nearest worthwhile speed we get a schedule that meets the new deadline constraint. The argument why this works is as follows: The distribution of task work remaining has by assumption not changed, but the deadline has effectively gotten shorter. Thus, all that has changed is the optimal value of K. This means the ratio of the new optimal speed to the old optimal speed is roughly the same for all points in the schedule, assuming that the function of energy versus speed has a reasonable shape.

4.6 Scheduling simultaneous tasks

When multiple tasks are ongoing, the ideal speed is not necessarily the sum of all the speeds for all those tasks. This is because power is not a linear function of speed, so superimposing schedules consumes a different amount of energy than running them sequentially. Unfortunately, computing a reasonable speed schedule that is the conjunction of two is extremely complex, so we avoid the issue by simply running at the maximum speed available when there are multiple tasks, and continue at that speed until no tasks remain.

Fortunately, in a mobile computer (and frequently in a desktop computer) there is only a single user and typically he will only notice the performance of the task with which he is currently actively involved. Therefore, typically there will be only one ongoing task at a time. Evidence supporting this comes from workload analyses we performed in [12] on months-long traces of eight desktop computers. We found that, depending on the user, between 94.7 and 99.3% of all user interface tasks finished before the next one began.

4.7 Scheduling no ongoing tasks

When no tasks are ongoing, nothing of importance is occurring, so the best speed to use is generally the minimum available. However, since our inference of tasks is imperfect, there may be ongoing tasks even when RightSpeed believes there are no such tasks. For instance, a task may have been triggered by a timeout instead of by a user interface event. We deal with this by reverting to a traditional interval-based scheduler when we know of no ongoing tasks. Such a scheduler divides time into intervals of some fixed length and chooses a speed for each interval based on the CPU utilization of recent past intervals. This way, if the CPU becomes busy from working on a task we cannot detect, the interval-based scheduler will nevertheless increase speed to deal with this unknown work.

One caveat is that when the number of known tasks becomes zero, recent past CPU utilization will likely be high because the system just finished working on a task. RightSpeed knows that this recent utilization is a poor predictor of future CPU utilization because it reflects a task that is no longer active. However, an interval-based scheduler has no knowledge of tasks, so it will interpret the high recent utilization as a sign that the next intervals will have high utilization. Accordingly, it will use an unnecessarily high CPU speed. To prevent this problem, when the number of known tasks becomes zero, RightSpeed waits for a short period of time at the minimum CPU speed before initiating the interval-based scheduler.

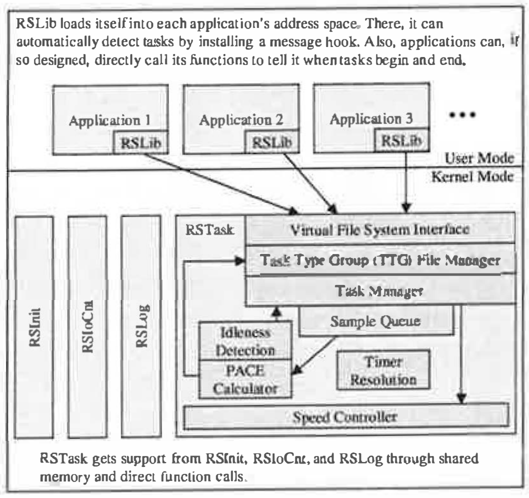

RSLib loads itself into each application's address space. There, it can automatically detect tasks by installing a message hook. Also, applications can, if so designed, directly call its functions to tell it when tasks begin and end.
Application 1
RSLib
Application 2
RSLib
Application 3
RSLib
...
User Mode
Kernel Mode
RSInit
RSIoCnt
RSLog
RSTask
Virtual File System Interface
Task Type Group (TTG) File Manager
Task Manager
Sample Queue
Idleness Detection
PACE Calculator
Timer Resolution
Speed Controller
RSTask gets support from RSInit, RSIoCnt, and RSLog through shared memory and direct function calls.

Figure 2: Architecture of RightSpeed

5 Implementation

In this section, we discuss how we implemented our approach on Windows 2000.

5.1 Architecture

Figure 2 shows the architecture of RightSpeed. The main component is RSTask, a kernel module that receives requests to begin and end tasks and schedules the CPU speed accordingly. Its main components are the speed controller, the task type group file manager, the automatic schedule computer, and the idleness detector, each of which we will discuss later. Alongside RSTask is RSIoCnt, a kernel module that interposes on all file system requests to monitor when any synchronous I/O's are ongoing. The next kernel component is RSLog, a low-overhead logger we use for benchmarking and debugging. The last kernel mode component is RSInit, a driver that starts before all other drivers and facilitates communication between them. In user mode we have RSLib, a user-level library that the system loads into the address space of every application. It interacts with the GUI to interpose the user interface event delivery system and thereby implement the automatic task detector. This library also exports functions that applications can use to communicate with RightSpeed.

5.2 Speed controller

The lowest-level component of RSTask is the speed controller. This component accepts requests to start and stop speed schedules and to transition to idle and maximum speed states. A speed schedule consists of a sequence of phases, each with a speed to use and a duration in multiples of the scheduling granularity. The speed controller internally handles any CPU-specific commands to change the speed. This modularity aided in porting RightSpeed to two different chips with different voltage scaling commands.

The scheduler also exports routines to pause and resume the current schedule when the CPU starts and stops waiting for I/O. A pause changes the speed to the minimum available. A resume determines how long the CPU spent waiting for I/O, accelerates the remaining part of the schedule accordingly, and resumes that schedule.

5.3 Timer resolution controller

The default timer resolution on Windows 2000 machines is about 10 ms. Our timer resolution controller reduces the timer resolution as much as possible using well-documented system calls [16]. On the systems we used, this makes timer resolution, and thus scheduling granularity, 1 ms.

5.4 Task type group file manager

Certain persistent information is associated with each task type: its deadline, its performance target, a sample of recent task CPU requirements, and a schedule to use for the next task. Thus, it makes sense to consider task types to be part of a virtual file system. We could have used one file per task type, but instead a file in this virtual file system is a *task type group file*, containing information about multiple related task types. A task type is uniquely identified by its file and its index within that file.

RSTask thus exposes a virtual file system interface consisting of these files. RSTask stores the information in these virtual files in real files in a reserved directory on the real file system, but RSTask exposes them as existing in the special directory \\.\RSTask. (The Unix analog would be /proc/rstask/.) Subdirectories of this directory are valid and supported; for instance, an application could choose to use a file called \\.\RSTask\AcmeCo\AcmeAppName\MyTasks.ttg. For performance, RSTask caches open files in memory and does not pass along changes to the copy to the on-disk file until the file is closed or until a global hourly timer goes off. (Users may change this period.)

Applications communicate with RSTask by performing I/O control requests on these virtual files. Supported control requests include beginning a task of a certain type and acquiring a task ID for it, ending the task with a given ID, changing the deadline for a task type, resetting the sample of recent work requirements for a task type, and various other minor ones. RSTask supports *fast I/O* control requests [15], a Windows 2000 optimization that speeds up I/O operations. As a further optimization, RSTask has a control request that ends one task and begins another; the automatic task detector in RSLib uses this to quickly signal the end of the previous user interface task when an application receives a new one.

5.5 Task manager and sample queue

RSTask keeps track of ongoing tasks and makes appropriate calls to the scheduler when tasks begin and end. Also, when a task ends, the task manager queues the information about how long this task took in the *sample queue*. It does not immediately invoke the PACE calculator since PACE calculation is best done when the CPU is otherwise idle.

As stated in Section 4.7, when no tasks are ongoing, we wait for a short period then initiate an interval-based scheduler. We do this on the Transmeta system in the following way. When RSTask detects the departure of the last ongoing task, it switches to the lowest available speed and sets a 50 ms timer. When the timer expires, it enters the LongRun™ automatic speed scheduling mode, which uses an interval-based strategy. We chose 50 ms because this is further backward than LongRun™'s scheduler ever looks. We have not yet implemented a scheme using an interval-based scheduler on the AMD system.

5.6 Idleness detector and automatic schedule computer

The idleness detector is another major component of RSTask. It is a thread running at priority 5, just above the idle level, so that it can easily detect when no important threads remain unblocked. If it is scheduled when an I/O is ongoing, it tells RSTask to pause the current schedule; RSIoCnt will later tell RSTask to resume the schedule when no synchronous I/O's remain in the system. If the idleness detector runs when no I/O is ongoing, it notifies RSTask that all tasks are complete. The other responsibility of the idleness detector is to invoke the PACE calculator on all unprocessed entries in the sample queue when the system is otherwise idle. Not only does this cause the overhead of PACE calculation to occur only when the system is idle, it also eliminates overhead due to saving and restoring floating-point state, as we will now describe.

The Windows 2000 kernel does not use floating-point instructions, so for performance reasons it does not save floating-point state when entering kernel mode or restore such state when leaving it. If we ran the PACE calculator in the kernel at arbitrary times, e.g., whenever a task completed, it would have to save and restore floating-point state to avoid corrupting the state of whatever thread it interrupted. By doing PACE calculation in the context of its own special thread, we make such save and restore operations unnecessary.

5.7 I/O counter

The other kernel module we will discuss is RSIoCnt. Its job is to count the pending synchronous I/O's and store this count in shared memory where the idleness detector can access it. It must also tell RSTask to resume any paused schedule whenever this count becomes zero.

We implemented RSIoCnt as a file system filter driver. A filter driver implements a filter device, a special kind of device extremely helpful in tracing system events in Windows NT/2000. A filter device can *attach* to an existing device, causing it to intercept any requests destined for that existing device. For more information about them, see [15, 10]. Our filter driver has low overhead because it merely counts the requests as they start and stop and passes them on.

Unfortunately, our approach limits one to filtering only non-network file systems. There are undocumented ways to filter network file systems and network devices, as shown in [10], but we do not do this in our prototype due to our stability goal.

5.8 User-mode library

We use a well-documented method to load RSLib, a user-mode library, into the address space of every process that makes GUI calls [14]. The main activity of this library is interposing on the delivery of user interface events to the application by using a *message hook* [14]. With this mechanism, we tell Windows to call a given function just before it successfully completes an application's request for the next message from the GUI.

RSLib also exports functions that applications can use. Most of these allow applications to specify task information. Applications can interact with RSTask without these calls, but they are helpful to application writers who prefer to use a function call interface rather than make I/O control calls to a virtual file system. RSLib exports other miscellaneous functions letting applications do things like disable automatic detection of their tasks.

6 Results

6.1 General overhead

In this subsection, we evaluate system overhead just from RightSpeed running unused in the background. There are two main sources of this overhead: (a) making the timer interrupt every 1 ms instead of every 10 ms

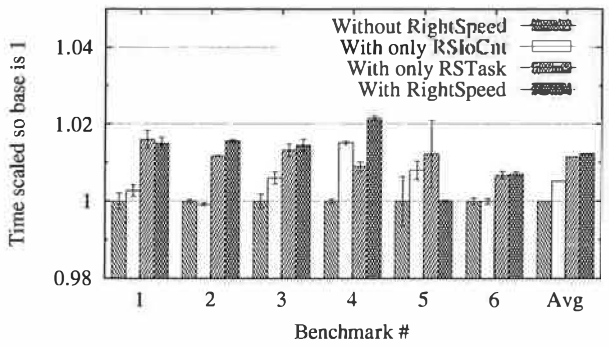


Figure 3: Time to perform various benchmarks without RightSpeed and with various components of RightSpeed enabled, shown with 95% confidence intervals. Note that the Y-axis origin is not zero.

causes interrupt-processing time to increase; and (b) filtering I/O requests to count them increases the time to perform each I/O.

To evaluate these effects, we ran the following benchmarks on a system with a 450 MHz Pentium III:

1. Read an uncached 32 KB file
2. Write a 100 KB file with write-through
3. Read 32 KB directly from the disk
4. Compile the RightSpeed logger device with the Windows DDK
5. Format a Ph.D. dissertation with LaTeX
6. Perform a CPU-intensive mathematical loop

We ran them without any RightSpeed modules loaded, with only the RSIoCnt module loaded, with only the RSTask module loaded, and with both of those two modules loaded. In all cases, we disabled the network to avoid interference from network interrupts. None of these benchmarks *use* RightSpeed at all; indeed, we did not even install RSLib to perform these experiments. We ran each benchmark enough times that the 95% confidence interval about the sample mean included no values more than 0.01% away from the sample mean, or 10,000 runs occurred, or 2,000 seconds passed, whichever came first. Figure 3 shows results.

We see that RSIoCnt adds 0.3–1.5% overhead, with an average of 0.5%, due to filtering I/O operations. If we did not have to use a file system filter to do this, e.g., if Microsoft provided hooks allowing one to simply count ongoing I/O's and be notified when the last I/O leaves the system, this overhead would likely be lower. We also observe that RSTask, by virtue of it reducing timer granularity from 10 ms to 1 ms, increases operation times by 0.7–1.6% with an average of 1.1%, presumably due to the system responding to more frequent timer interrupts. Combined, the overhead is 1.2% on average.

Operation	Time
Load and initialize RSLib for a process	1.401 ms
Install message hook	8.532 μs
Open system auto task type group file	159.777 μs
Get application name	12.583 μs
Open per-app auto task type group file	121.229 μs
Intercept non-user-interface message	2.265 μs
Intercept & handle user interface message	7.605 μs
Evaluate message type	1.013 μs
End task and begin another	3.575 μs
Simple I/O control request to RSTask	1.162 μs
Begin a task	3.450 μs
Kernel-mode component	1.345 μs
End a task	2.530 μs
Kernel-mode component	1.134 μs
End one task and begin another	3.462 μs
Kernel-mode component	1.441 μs

Table 3: Average time RightSpeed takes to perform common operations on the AMD machine at 900 MHz

6.2 Time to perform RightSpeed operations

The next set of results evaluates the time to perform various RightSpeed operations. We performed these measurements on the AMD system, since accurately evaluating performance on the Transmeta system is difficult for two reasons: (a) the dynamic translation of code the chip performs can cause large differences from one run to another, and (b) confidentiality agreements preclude us from publishing certain measurements of the prototype system. In all cases speed changing was disabled to not confound the measurement of durations, so all runs are at 900 MHz. Most of these results we measured directly by making an entry in the log each time an operation started or stopped. However, some of them, such as intercepting a message, involve hidden overhead, so we measured them by running with and without the operation and subtracting. We ran each operation 10,100 times and discarded the first 100. Table 3 shows the mean results.

We see that the overhead of linking RSLib into each application is about 1.4 ms; this occurs only once per application, when it starts. Some of this is RSLib's initialization, including installing the message hook and opening the automatic task type group files, but this accounts for little of it. In these benchmarks, the application task type group file is in the file cache, but even if it were not the time to load would not be significantly more.

The overhead of hooking all messages delivered to applications is also small. For non-user-interface messages, the overhead is 2.3 μs per message. For user interface messages, the message hook must determine the event type and communicate that this task is beginning and the previous task is ending to RightSpeed. The total extra time is small, approximately 7.6 μs per message. Considering that messages arrive on the order of every few milliseconds, and that user interface messages arrive even less often (at worst about every 14 ms in the case of rapid mouse movement, and more typically about once every 150 ms if the user is typing at 40 words per minute), the total overhead is low.

RightSpeed operation microbenchmarks show more detail about the cause of overhead. Each I/O control request takes about 1–2 μs due to the time to trap into kernel mode and to check and copy data from user buffers to kernel buffers. Inside RSTask, the time to begin a task is about 1.3 μs and the time to end a task is about 1.1 μs. The most common operation, beginning one task and ending another that is already considered complete, takes about 1.4 μs of kernel time. Note that this is less than the sum of the time to begin a task and to end a task because of various optimizations for this case. For example, we look up the task type group file only once and we acquire and release the spin lock controlling access to the ongoing tasks list only once.

6.3 Effect on performance

Applications can specify performance targets for tasks. However, since Windows 2000 is not a real-time operating system, scheduling decisions do not necessarily happen precisely when they should, so RightSpeed will not necessarily meet these targets. In this subsection, we evaluate how closely it does.

These evaluations require workloads. We derived these workloads from traces of users performing their normal business on desktop machines running Windows NT or Windows 2000. For more details on the tracing, see [10]. Each workload corresponds to all tasks requiring no I/O that were triggered by keystroke and mouse click events delivered to a particular application for a particular user during the several months that user was traced. Table 4 gives a brief description of each workload; for more details about the users and applications, see [12] and [11]. We inferred when tasks began and ended using the method from [12].

Our traces do not give us sufficient information to precisely recreate the workloads. For instance, we do not collect disk contents and we irreversibly encrypt alphanumeric keystrokes. To simulate RightSpeed, however, we need only know when and for how long each task ran. Thus, we use a simulator that simulates each task by performing additions repeatedly in a tight loop for the same number of cycles as the original task took. Our workload simulator indicates the beginning of each task to RightSpeed with an explicit RSLib call; it sleeps

Workload	User	Application	Key events	Click events
1	1	explorer	17,105	9,549
2	2	explorer	27,972	19,866
3	3	explorer	9,905	2,617
4	4	explorer	11,276	6,301
5	5	explorer	21,297	9,291
6	6	explorer	6,096	5,381
7	7	explorer	6,938	4,443
8	8	explorer	24,208	9,337
9	1	netscape	797,642	22,512
10	2	iexplore	193,823	59,667
11	3	psp	64,229	3,320
12	4	outlook	359,839	14,984
13	5	outlook	109,202	2,633
14	6	grpwise	275,972	13,576
15	7	winword	50,799	2,766
16	8	excel	13,891	2,016

Table 4: Traced application workloads we use in certain simulations

for 2 ms at the end of each task to let RightSpeed automatically detect the end of the task. We run this workload simulator on the AMD machine to measure the performance obtained when RightSpeed schedules the speeds.

We evaluate RightSpeed's performance as follows. We assign performance targets corresponding to an average pre-deadline speed of 630 MHz for keystroke tasks and 765 MHz for mouse click tasks. For each workload, we calculate how many deadlines it theoretically should miss and how much total delay past deadlines it should achieve. We then simulate RightSpeed to see how many deadlines it actually misses and the total delay it actually achieves.

We find that RightSpeed misses 1.5% fewer to 0.3% more deadlines than the target, with an average absolute error of 0.4%. It has delay from 0.5% less to 0.1% more than the target with an average absolute error of 0.2%. Since RightSpeed conservatively rounds speeds for intervals to maximize the probability of making deadlines, it is not surprising that it tends to miss fewer deadlines and have less delay than the target. Nevertheless, the absolute error is very low, showing that RightSpeed is effective at meeting performance targets even though it must use the millisecond-granularity timer of Windows 2000 and even though Windows 2000 makes no guarantees about when speed-changing routines will actually execute.

6.4 Time to perform PACE calculations

We also measured the average time to perform PACE calculations for tasks. We performed this experiment for the user 1 workload running explorer. We found that adding the sample value to the task type group information and recomputing the schedule accordingly took an average of 4.447 μs $\pm$ 0.312 μs, the 95% confidence interval. (The standard deviation is very high, 143.633 μs, because occasionally PACE calculations are interrupted by a context switch and take milliseconds instead of microseconds to complete.) Note that these calculations were always performed at the slowest, 500 MHz setting. So, we see that PACE calculations can be made quite quickly given all our optimizations.

6.5 Effect of overhead on energy consumption

To evaluate the effect of RightSpeed overhead on energy consumption, we ran some workloads on the Transmeta machine both with and without RightSpeed. To equalize performance, we instructed RightSpeed to not use the PACE calculator but instead use an algorithm identical to Transmeta's LongRun^{TM} strategy. Table 5 shows the results for five short workloads derived from VTrace traces. We observe that the performance characteristics (deadlines missed and total delay) of RightSpeed mimicking LongRun^{TM} are very close to that of LongRun^{TM} by itself, so it is meaningful to directly compare the energy consumption of the two. We see that simulating LongRun^{TM} with RightSpeed has little effect on the total energy consumption. In other words, the overhead of signaling the beginnings and ends of tasks, and of implementing the speed schedule in software instead of hardware is insignificant.

6.6 Effect of PACE on future processors

Because the real processors on which we implemented RightSpeed derive little efficiency from using one setting versus another, PACE cannot save sufficient energy on them to make its implementation worthwhile. To evaluate the effectiveness of our PACE calculator, in this section we conduct simulations assuming future processors with better DVS characteristics. Our simulations differ from those in [11] since we do not make the same assumptions about scheduling capabilities. In particular, we consider a finite number of settings and limited timer granularity.

For our simulations, we consider three processors, each with a minimum setting running at 200 MHz and consuming 1 W, and each with power consumption proportional to speed cubed. (This cubic relationship assumes either a very low threshold voltage or a threshold voltage that is varied proportionally to supply voltage using technology like that in [8].) The three processors differ only in their maximum speeds: 600 MHz, 800 MHz,

Workload	Without RightSpeed			With RightSpeed mimicking LongRun™			Relative energy increase
	Deadlines missed	Delay	Energy	Deadlines missed	Delay	Energy	
A	21 out of 30,635	0.947 s	172.8 J	21 out of 30,635	0.940 s	174.0 J	+0.7%
B	8 out of 19,310	1.172 s	94.58 J	8 out of 19,310	1.172 s	94.86 J	+0.2%
C	1,792 out of 32,288	118.833 s	1007 J	1,796 out of 32,288	118.860 s	1006 J	-0.1%
D	61 out of 19,770	2.364 s	126.8 J	61 out of 19,770	2.366 s	126.8 J	0.0%
E	99 out of 20,641	4.070 s	379.1 J	99 out of 20,641	4.061 s	376.1 J	-0.8%

Table 5: Comparison of using built-in LongRun™ scheduling versus doing this scheduling with RightSpeed

and 1 GHz. We assume the processors can only run at multiples of 50 MHz and the timer granularity is 0.1 ms.

Since our simulations occur on virtual hardware, we can run them much faster than real time. So, we can use longer workloads than those in Table 4, which were restricted to a single application. Instead, we use eight workloads, each corresponding to *all* activity of a traced user.

All the algorithms we simulate, except for the no-DVS algorithm, will use the same performance target, so that we can compare them fairly using only energy consumption. The performance target is to have an average pre-deadline speed of 400 MHz and a post-deadline speed of 600 MHz. The four algorithms we consider are:

- **Flat.** The pre-deadline speed is constant.
- **Stepped.** The pre-deadline speed begins at 200 MHz and is incremented by 50 MHz after each interval. Interval length is chosen to achieve the desired average pre-deadline speed. This models algorithms such as that used by Transmeta's LongRun™ [7].
- **Past/Peg.** The pre-deadline speed is constant at 200 MHz for the first interval, then is pegged to the maximum. Interval length is chosen to achieve the desired average pre-deadline speed. This models the algorithm suggested in [5].
- **PACE.** The pre-deadline speed schedule is computed by PACE using an estimate of task work distribution derived from the most recent tasks of the same type.

Results are in Table 6 and summarized in Figure 4. Note that Flat does not change its behavior for different maximum speeds, so we present its results only for a 600 MHz maximum speed.

One interesting observation is that the greater the range of speeds available on the CPU, the more energy efficient the Stepped and PACE algorithms become. For example, per-task average CPU energy consumption under PACE decreases 19.5% when switching from a CPU with maximum speed 600 MHz to one with maximum speed 1 GHz. This is because the availability of a higher

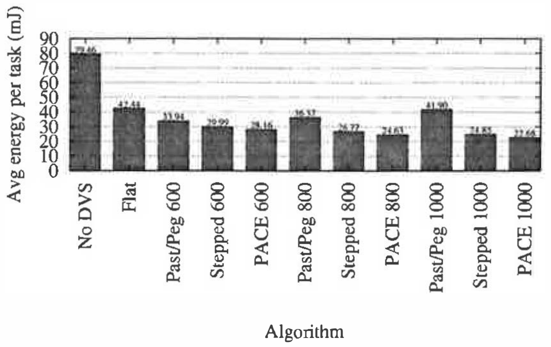


Figure 4: Summary of Table 6, showing average per-task energy consumption averaged over all workloads for various algorithms. Numbers after an algorithm identify the maximum CPU speed made available to that algorithm.

speed on the CPU allows a schedule to begin a task running more slowly, since it can more easily make up for this slowness by running even faster later in the schedule. The ability to run slowly at the beginning saves energy in the common case where the task requires little work, since the schedule never proceeds past the low-energy beginning part. PACE takes advantage of the broader range of speeds to find a better schedule, while Stepped just happens to work better with the larger set of speeds. Past/Peg, on the other hand, does worse with a greater range of speeds. Essentially, Past/Peg ignores all but the two extreme settings of the CPU, and we see that this is costly in terms of energy consumption; we conclude that using intermediate speeds can save energy.

We also see from these results that PACE is always the best algorithm, followed by Stepped, followed by Past/Peg, followed by Flat. This echoes the results from [12], and shows that even when we require PACE to deal with limited settings and timer granularity, it is still an improvement over existing DVS algorithms.

Furthermore, we predicted in Section 3 that the greater the available CPU speed range, the better PACE would do in comparison to other algorithms, and we see this borne out in our simulation results. On the CPU with maximum speed 600 MHz, PACE reduces energy con-

	Maximum speed 600 MHz					*Maximum speed 800 MHz*			*Maximum speed 1 GHz*		
User	**No DVS**	**Flat**	**P/Peg**	**Stepped**	**PACE**	**P/Peg**	**Stepped**	**PACE**	**P/Peg**	**Stepped**	**PACE**
1	44.83	23.29	16.11	14.80	13.67	15.85	13.29	11.87	16.90	12.38	10.92
2	112.36	67.00	64.62	57.07	53.60	69.44	49.37	45.59	81.19	44.75	40.93
3	81.93	39.62	31.19	25.25	23.34	35.78	23.80	21.05	42.44	22.93	20.06
4	48.07	22.20	8.44	9.04	7.78	8.90	8.66	7.39	9.75	8.44	7.18
5	80.24	41.70	25.45	24.59	23.43	25.76	21.86	20.70	28.26	20.23	19.13
6	51.20	23.47	12.02	11.12	10.09	11.71	10.80	9.39	12.39	10.61	9.17
7	132.34	77.22	73.37	64.29	61.26	78.65	55.99	52.48	91.16	51.00	47.47
8	84.75	45.02	40.32	33.74	32.10	44.84	30.43	28.55	53.09	28.44	26.61
Avg	*79.46*	*42.44*	*33.94*	*29.99*	*28.16*	*36.37*	*26.77*	*24.63*	*41.90*	*24.85*	*22.68*

Table 6: Simulation results showing average per-task energy consumption, in mJ, for various algorithms, workloads, and maximum CPU speeds. All algorithms except "No DVS" achieve the same performance target by using a 400 MHz average pre-deadline speed and a 600 MHz constant post-deadline speed. P/Peg stands for Past/Peg.

sumption by 6.1% compared to Stepped; with maximum speed 800 MHz, the reduction is 8.0%; with maximum speed 1 GHz, the reduction is 8.7%.

In conclusion, we find that even when a finite set of speeds are available and the timer granularity is limited, PACE is still an improvement over other algorithms. We find that having higher speeds available on the CPU helps PACE reduce energy consumption, and furthermore PACE does better the greater the range of speeds available on the CPU. This is an important lesson for chip designers, who may think that providing the capability of running at high voltages and therefore high speeds will increase energy consumption. We see here that with proper energy management using PACE, provision of higher speeds can actually *reduce* energy consumption.

7 Future work

7.1 Modifying applications

An important next step in this research is to insert calls to RightSpeed into various applications, such as movie players, to communicate task information to RightSpeed. We have shown that RightSpeed is good at meeting deadline targets, and this will pay off better once we modify applications in this manner.

7.2 User testing

In this paper, we have relied on user interface studies that suggest a connection between making deadlines and user-perceived response time instead of conducting user experiments ourselves. It will be important in future work to make sure that the performance targets RightSpeed assigns to automatically detected tasks ensure a satisfactory user experience.

7.3 PACE calculator

We hope in future to test the PACE calculator on a real system with a large range of worthwhile settings to evaluate its actual effect on the energy consumption of such a system.

7.4 Specification of performance targets

For some applications, the best way to specify the performance target may not be an average pre-deadline speed or an equivalent DVS algorithm, but rather a target fraction of deadlines to make, e.g., to say that 99% of tasks should complete by their deadline. In future, we would like to devise a way for RightSpeed to meet this kind of performance target with high accuracy and energy efficiency.

7.5 Predicting I/O

Our approach to dealing with I/O is somewhat unsatisfactory, as we do not consider the I/O time a task requires until after it actually occurs. A better approach would be to model the probability distribution of task I/O requirements for each task type and use this distribution to compute a more optimal schedule at the outset of the task. This requires a more complicated model of speed and voltage scheduling, and consequently a more complicated solution to computing an optimal schedule than PACE currently uses.

8 Summary and Conclusions

We implemented RightSpeed, a task-based speed and voltage scheduler for systems running Windows 2000 on Transmeta or AMD processors. Unlike traditional DVS schedulers, which use interval-based methods to change speed merely according to recent CPU usage, RightSpeed considers tasks and their performance constraints. RightSpeed is an improvement over other task-based schedulers since it uses PACE to compute optimal speed schedules and uses an efficient heuristic to automatically detect tasks triggered by user interface events. RightSpeed also distinguishes itself by running on Windows, the most popular laptop operating system.

RightSpeed obtains task information in two ways. First, applications can directly indicate when tasks begin and end, what type of task each task is, and the performance targets for each task type. Second, RightSpeed uses an *automatic task detector* to infer task information for applications not using the RightSpeed task specification interface. This detector infers that a task begins whenever it observes a user interface event such as a keystroke.

RightSpeed also features a *PACE calculator*. This allows RightSpeed to automatically monitor the work requirements of tasks as they complete, deduce a probability distribution of work requirements for each task type, and from those to compute optimal schedules for scheduling CPU speed when tasks of those type run. It computes these schedules using the theory of PACE, described in [11].

We demonstrated that RightSpeed can meet performance targets applications specify, despite the fact that Windows 2000 does not provide scheduling guarantees. We also demonstrated that the overhead due to using RightSpeed is small. The overhead due to low-level system modifications, including monitoring I/O and increasing timer resolution, is only 1.2% on average. The overhead due to other aspects of RightSpeed is also modest, on the order of a few microseconds to perform most operations. Even PACE calculation, involving complicated floating-point operations, takes only about 4.4 μs per task on a 500 MHz processor, thanks to several optimizations.

The systems to which we ported RightSpeed have DVS characteristics quite different from the idealized conditions given in [11], since they have limited scheduling granularity, a limited supply of speeds, and a nonlinear relationship between speed squared and energy. We therefore developed techniques to apply PACE to such real systems, and implemented them in RightSpeed.

Unfortunately, these processors derive so little efficiency from using one setting versus another that actual savings from the PACE calculator are minuscule. One system even contains settings that are never worthwhile for PACE schedules to use. This departure from the theoretical model may result from overly conservative speed/voltage settings from the chip manufacturer or poor circuit engineering, or it may reflect problems with voltage scaling not reflected in the standard theoretical model. In any case, assuming that the problems are with the chips and not with the model, we therefore performed simulations on theoretical processors whose settings' efficiencies more closely match those expected from semiconductor theory.

We found in our simulations that our version of PACE, optimized for speed and modified to take into account limits of speed and time granularity on real systems, saves energy compared to other algorithms. Furthermore, PACE is most effective at improving algorithms when the CPU has a large speed range. PACE reduces energy consumption compared to the Stepped algorithm by 6.1% when the speed range is 200 MHz–600 MHz; this relative improvement rises to 8.7% when the speed range expands to 200 MHz–1 GHz.

We also found that as long as one uses the PACE algorithm, CPU energy decreases when the range of available speeds increases. For example, on a CPU with a speed range of 200 MHz–1 GHz, we consume 19.5% less energy than on a CPU with a speed range of only 200 MHz–600 MHz. An important lesson from this is that the current practice of reducing the maximum speed of processors marketed for mobile environments may be misguided. Providing the ability to run at a high speed, even if it can only be for a short time due to thermal constraints, can not only make a processor more attractive to consumers evaluating them in terms of their maximum performance, but can also actually reduce energy consumption by providing DVS algorithms with more options. To take advantage of these options, however, the system needs to use an algorithm like PACE that only uses high speeds when necessary.

The code for RightSpeed is available on the World Wide Web at http://www.cs.berkeley.edu/~lorch/rightspeed/. Although Jacob Lorch is currently affiliated with Microsoft, he performed all implementation work while still a student at UC Berkeley. Thus, the implementation used no internal Microsoft knowledge or documentation.

9 Acknowledgments

This material is based upon work supported by the State of California MICRO program, AT&T Laboratories, Cisco Systems, Fujitsu Microelectronics, IBM, Intel Corporation, Maxtor Corporation, Microsoft Corporation, Sony Research Laboratories, Sun Microsystems, Toshiba Corporation, and Veritas Software.

This work would not have been possible without the cooperation of people from both Transmeta and AMD, who provided not only hardware but also detailed guidance in using it. Those people include Marc Fleischmann and Rowan Hamilton from Transmeta and Richard Russell, Dave Tobias, and Fred Weber from AMD. Also, we offer great thanks to the many users of our tracer whose traces yielded the workloads for this paper. Finally, we thank the anonymous reviewers of this paper and especially our shepherd, Deborah Wallach, for their many helpful comments and suggestions.

References

[1] AMD. Mobile AMD Athlon 4 processor model 6 CPGA data sheet. On the World Wide Web at http://www.amd.com/products/cpg/athlon/techdocs/pdf/24319.pdf, August 2001.

[2] E. Chan, K. Govil, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the First ACM International Conference on Mobile Computing and Networking (MOBICOM 95)*, pages 13–25, November 1995.

[3] K. Flautner and T. Mudge. Vertigo: automatic performance-setting for Linux. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 105–116, December 2002.

[4] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the Seventh ACM International Conference on Mobile Computing and Networking (MOBICOM 2001)*, July 2001.

[5] D. Grunwald, P. Levis, K. I. Farkas, C. B. Morrey III, and M. Neufeld. Policies for dynamic clock scheduling. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, October 2000.

[6] I. Hong, M. Potkonjak, and M. B. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processor. In *Proceedings of the International Conference on Computer Aided Design*, pages 653–656, November 1998.

[7] A. Klaiber. The technology behind Crusoe™ processors. White paper, Transmeta Corporation, January 2000.

[8] T. Kuroda, K. Suzuki, S. Mita, T. Fujita, F. Yamane, F. Sano, A. Chiba, Y. Watanabe, K. Matsuda, T. Maeda, T. Sakurai, and T. Furuyama. Variable supply-voltage scheme for low-power high-speed CMOS digital design. *IEEE Journal of Solid-State Circuits*, 33(3):454–462, March 1998.

[9] J. R. Lorch. *Operating Systems Techniques for Reducing Processor Energy Consumption.* PhD thesis, Computer Science Division, EECS Department, University of California at Berkeley, 2001. Available online at http://www.cs.berkeley.edu/~lorch/papers/.

[10] J. R. Lorch and A. J. Smith. The VTrace tool: building a system tracer for Windows NT and Windows 2000. *MSDN Magazine*, 15(10):86–102, October 2000.

[11] J. R. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE. In *Proceedings of the 2001 ACM SIGMETRICS Conference*, pages 50–61, June 2001.

[12] J. R. Lorch and A. J. Smith. Using user interface event information in dynamic voltage scaling algorithms. Technical Report UCB/CSD-02-1190, Computer Science Division, EECS, University of California at Berkeley, August 2002.

[13] I. S. MacKenzie and C. Ware. Lag as a determinant of human performance in interactive systems. In *Proceedings of INTERCHI '93*, pages 24–29, April 1993.

[14] Microsoft Corporation. *Platform SDK Documentation*, 2000.

[15] R. Nagar. *Windows NT File System Internals.* O'Reilly and Associates, Inc., Sebastopol, CA, 1997.

[16] G. Nebbett. *Windows NT/2000 Native API Reference.* Macmillan Technical Publishing, Indianapolis, IN, 2000.

[17] T. Pering, T. Burd, and R. W. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 76–81, August 1998.

[18] T. Pering, T. Burd, and R. W. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, pages 96–101, July 2000.

[19] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 89–102, October 2001.

[20] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley, Reading, MA, 1998.

[21] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 13–23, November 1994.

[22] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley, Reading, MA, 1993.

[23] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of the IEEE 36th Annual Symposium on Foundations of Computer Science*, pages 374–382, October 1995.

Energy Aware Lossless Data Compression

Kenneth Barr and Krste Asanović

MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139
E-mail: {kbarr,krste}@lcs.mit.edu

Abstract

Wireless transmission of a bit can require over 1000 times more energy than a single 32-bit computation. It would therefore seem desirable to perform significant computation to reduce the number of bits transmitted. If the energy required to compress data is less than the energy required to send it, there is a net energy savings and consequently, a longer battery life for portable computers. This paper reports on the energy of lossless data compressors as measured on a StrongARM SA-110 system. We show that with several typical compression tools, there is a net energy *increase* when compression is applied before transmission. Reasons for this increase are explained, and hardware-aware programming optimizations are demonstrated. When applied to Unix *compress*, these optimizations improve energy efficiency by 51%. We also explore the fact that, for many usage models, compression and decompression need not be performed by the same algorithm. By choosing the lowest-energy compressor and decompressor on the test platform, rather than using default levels of compression, overall energy to send compressible web data can be reduced 31%. Energy to send harder-to-compress English text can be reduced 57%. Compared with a system using a single optimized application for both compression and decompression, the asymmetric scheme saves 11% or 12% of the total energy depending on the dataset.

1 Introduction

Wireless communication is an essential component of mobile computing, but the energy required for transmission of a single bit has been measured to be over 1000 times greater than a single 32-bit computation. Thus, if 1000 computation operations can compress data by even one bit, energy should be saved. However, accessing memory can be over 200 times more costly than computation on our test platform, and it is memory access that dominates most lossless data compression algorithms. In fact, even moderate compression (e.g. `gzip -6`) can require so many memory accesses that one observes an *increase* in the overall energy required to send certain data.

While some types of data (e.g., audio and video) may accept some degradation in quality, other data must be transmitted faithfully with no loss of information. Fidelity can not be sacrificed to reduce energy as is done in related work on lossy compression. Fortunately, an understanding of a program's behavior and the energy required by major hardware components can be used to reduce energy. The ability to efficiently perform efficient lossless compression also provides second-order benefits such as reduction in packet loss and less contention for the fixed wireless bandwidth. Concretely, if n bits have been compressed to m bits ($n > m$); c is the cost of compression and decompression; and w is the cost per bit of transmission and reception; compression is energy efficient if $\frac{c}{n-m} < w$. This paper examines the elements of this inequality and their relationships.

We measure the energy requirements of several lossless data compression schemes using the "Skiff" platform developed by Compaq Cambridge Research Labs. The Skiff is a StrongARM-based system designed with energy measurement in mind. Energy usage for CPU, memory, network card, and peripherals can be measured individually. The platform is similar to the popular Compaq iPAQ handheld computer, so the results are relevant to handheld hardware and developers of embedded software. Several families of compression algorithms are analyzed and characterized, and it is shown that carelessly applying compression prior to transmission may cause an overall energy increase. Behaviors and resource-usage patterns are highlighted which allow for energy-efficient lossless compression of data by applications or network drivers. We focus on situations in which the mixture of high energy network operations and low energy processor operations can be adjusted so that overall energy is lower. This is possible even if the number of total operations, or time to complete them, increases. Finally, a new energy-aware data compression strategy composed of an asymmetric compressor and decompressor is presented and measured.

Section 2 describes the experimental setup including equipment, workloads, and the choice of compression applications. Section 3 begins with the measurement of an encouraging communication-computation gap, but shows that modern compression tools do not exploit the the low relative energy of computation versus communication. Factors which limit energy reduction are presented. Section 4 applies an understanding of these factors to reduce overall energy of transmission though hardware-conscious optimizations and asymmetric compression choices. Section 5 discusses related work, and Section 6 concludes.

2 Experimental setup

While simulators may be tuned to provide reasonably accurate estimations of a particular system's energy, observing real hardware ensures that complex interactions of components are not overlooked or oversimplified. This section gives a brief description of our hardware and software platform, the measurement methodology, and benchmarks.

2.1 Equipment

The Compaq Personal Server, codenamed "Skiff," is essentially an initial, "spread-out" version of the Compaq iPAQ built for research purposes [13]. Powered by a 233 MHz StrongARM SA-110 [29, 17], the Skiff is computationally similar to the popular Compaq iPAQ handheld (an SA-1110 [18] based device). For wireless networking, we add a five volt Enterasys 802.11b wireless network card (part number CSIBD-AA). The Skiff has 32 MB of DRAM, support for the Universal Serial Bus, a RS232 Serial Port, Ethernet, two Cardbus sockets, and a variety of general purpose I/O. The Skiff PCB boasts separate power planes for its CPU, memory and memory controller, and other peripherals allowing each to be measured in isolation (Figure 1). With a Cardbus extender card, one can isolate the power used by a wireless network card as well. A programmable multimeter and sense resistor provide a convenient way to examine energy in a active system with error less than 5% [47].

The Skiff runs ARM/Linux 2.4.2-rmk1-np1-hh2 with PCMCIA Card Services 3.1.24. The Skiff has only 4 MB of non-volatile flash memory to contain a file system, so the root filesystem is mounted via NFS using the wired ethernet port. For benchmarks which require file system access, the executable and input dataset is brought into RAM before timing begins. This is verified by observing

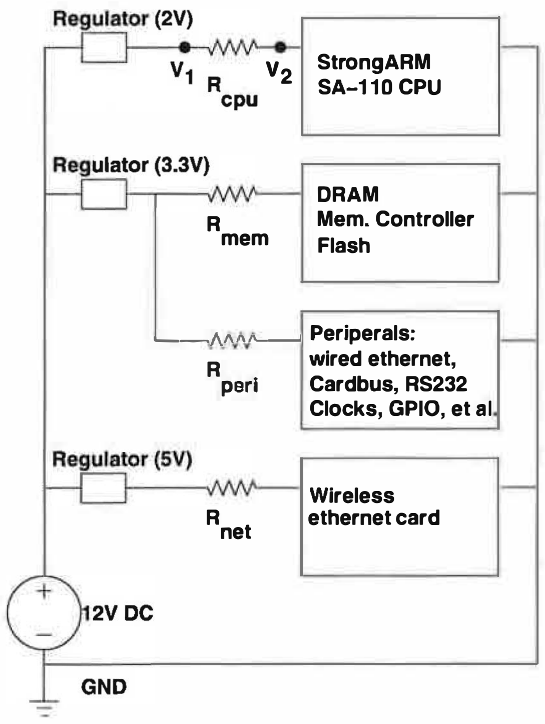


Figure 1. Simplified Skiff power schematic

the cessation of traffic on the network once the program completes loading. I/O is conducted in memory using a modified SPEC harness [42] to avoid the large cost of accessing the network filesystem.

2.2 Benchmarks

Figure 2 shows the performance of several lossless data compression applications using metrics of compression ratio, execution time, and static memory allocation. The datasets are the first megabyte (English books and a bibliography) from the Calgary Corpus [5] and one megabyte of easily compressible web data (mostly HTML, Javascript, and CSS) obtained from the homepages of the Internet's most popular websites [32, 25]. Graphics were omitted as they are usually in compressed form already and can be recognized by application-layer software via their file extensions. Most popular repositories ([4, 10, 11]) for comparison of data compression do not examine the memory footprint required for compression or decompression. Though static memory usage may not always reflect the size of the application's working set, it is an essential consideration in mobile computing where memory is a more precious resource. A detailed look at the memory used by each application, and its effect on time, compression ratio, and energy will be presented in Section 3.3.

Figure 2 confirms that we have chosen an array of ap-

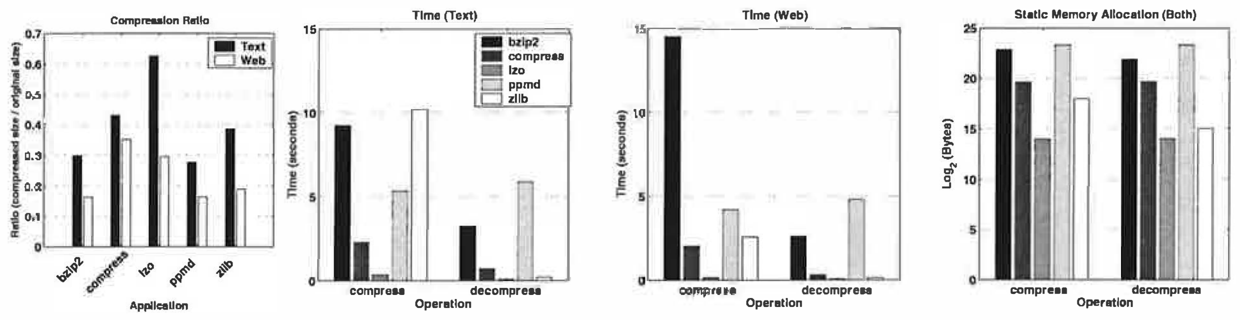


Figure 2. Benchmark comparison by traditional metrics

plications that span a range of compression ratios and execution times. Each application represents a different family of compression algorithms as noted in Table 1. Consideration was also given to popularity and documentation, as well as quality, parameterizability, and portability of the source code. The table includes the default parameters used with each program. To avoid unduly handicapping any algorithm, it is important to work with well-implemented code. Mature applications such as *compress*, *bzip2*, and *zlib* reflect a series of optimizations that have been applied since their introduction. While *PPMd* is an experimental program, it is effectively an optimization of the Prediction by Partial Match (PPM) compressors that came before it. *LZO* represents an approach for achieving great speed with LZ77. Each of the five applications is summarized below assuming some familiarity with each algorithm. A more complete treatment with citations may be found in [36].

zlib combines LZ77 and Huffman coding to form an algorithm known as "deflate." The LZ77 sliding window size and hash table memory size may be set by the user. LZ77 tries to replace a string of symbols with a pointer to the longest prefix match previously encountered. A larger window improves the ability to find such a match. More memory allows for less collisions in the *zlib* hash table. Users may also set an "effort" parameter which dictates how hard the compressor should try to extend matches it finds in its history buffer. *zlib* is the library form of the popular *gzip* utility (the library form was chosen as it provides more options for trading off memory and performance). Unless specified, it is configured with parameters similar to *gzip*.

LZO is a compression library meant for "real-time" compression. Like *zlib*, it uses LZ77 with a hash table to perform searches. *LZO* is unique in that its hash table can be sized to fit in 16KB of memory so it can remain in cache. Its small footprint, coding style (it is written completely with macros to avoid function call overhead), and ability to read and write data "in-place" without additional copies make *LZO* extremely fast. In the interest of speed, its hash table can only store pointers to 4096 matches, and no effort is made to find the longest match. Match length and offset are encoded more simply than in *zlib*.

compress is a popular Unix utility. It implements the LZW algorithm with codewords beginning at nine bits. Though a bit is wasted for each single 8-bit character, once longer strings have been seen, they may be replaced with short codes. When all nine-bit codes have been used, the codebook size is doubled and the use of ten-bit codes begins. This doubling continues until codes are sixteen bits long. The dictionary becomes static once it is entirely full. Whenever *compress* detects decreasing compression ratio, the dictionary is cleared and the process beings anew. Dictionary entries are stored in a hash table. Hashing allows an average constant-time access to any entry, but has the disadvantage of poor spatial locality when combining multiple entries to form a string. Despite the random dispersal of codes to the table, common strings may benefit from temporal locality. To reduce collisions, the table should be sparsely filled which results in wasted memory. During decompression, each table entry may be inserted without collision.

PPMd is a recent implementation of the PPM algorithm. Windows users may unknowingly be using *PPMd* as it is the text compression engine in the popular *WinRAR* program. PPM takes advantage of the fact that the occurrence of a certain symbol can be highly dependent on its context (the string of symbols which preceded it). The PPM scheme maintains such context information to estimate the probability of the next input symbol to appear. An arithmetic coder uses this stream of probabilities to efficiently code the source. As the model becomes more accurate, the occurrence of a highly likely symbol requires fewer bits to encode. Clearly, longer contexts will improve the probability estimation, but it requires time to amass large contexts (this is similar to the startup effect in LZ78). To account for this, "escape symbols" exist to progressively step down to shorter context lengths. This introduces a trade-off in which encoding a long series of escape symbols can require more space than is saved by the use of large contexts. Stor-

Application (Version)	Source	Algorithm	Notes (defaults)
bzip2 (0.1pl2)	[37]	BWT	RLE→BWT→MTF→RLE→HUFF (900k block size)
compress (4.0)	[21]	LZW	modified Unix Compress based on Spec95 (16 bit codes (maximum))
LZO (1.07)	[33]	LZ77	Favors speed over compression (lzo1x_12. 4K entry hash table uses 16KB)
PPMd (variant I)	[40]	PPM	used in "rar" compressor (Order 4, 10MB memory, restart model)
zlib (1.1.4)	[9]	LZ77	library form of gzip (Chaining level 6 / 32K Window / 32K Hash Table)

Table 1. Compression applications and their algorithms

ing and searching through each context accounts for the large memory requirements of PPM schemes. The length of the maximum context can be varied by *PPMd*, but defaults to four. When the context tree fills up, *PPMd* can clear and start from scratch, freeze the model and continue statically, or prune sections of the tree until the model fits into memory.

bzip2 is based on the Burrows Wheeler Transform (BWT) [8]. The BWT converts a block S of length n into a pair consisting of a permutation of S (call it L) and an integer in the interval $[0..n-1]$. More important than the details of the transformation is its effect. The transform collects groups of identical input symbols such that the probability of finding a symbol s in a region of L is very high if another instance of s is nearby. Such an L can be processed with a "move-to-front" coder which will yield a series consisting of a small alphabet: runs of zeros punctuated with low numbers which in turn can be processed with a Huffman or Arithmetic coder. For processing efficiency, long runs can be filtered with a run length encoder. As block size is increased, compression ratio improves. Diminishing returns (with English text) do not occur until block size reaches several tens of megabytes. Unlike the other algorithms, one could consider BWT to take advantage of symbols which appear in the *"future"*, not just those that have passed. *bzip2* reads in blocks of data, run-length-encoding them to improve sort speed. It then applies the BWT and uses a variant of move-to-front coding to produce a compressible stream. Though the alphabet may be large, codes are only created for symbols in use. This stream is run-length encoded to remove any long runs of zeros. Finally Huffman encoding is applied. To speed sorting, *bzip2* applies a modified quicksort which has memory requirements over five times the size of the block.

2.3 Performance and implementation concerns

A compression algorithm may be implemented with many different, yet reasonable, data structures (including binary tree, splay tree, trie, hash table, and list) and yield vastly different performance results [3]. The quality and applicability of the implementation is as important as the underlying algorithm. This section has presented implementations from each algorithmic family. By choosing a top representative in each family, the implementation playing field is leveled, making it easier to gain insight into the underlying algorithm and its influence on energy. Nevertheless, it is likely that each application can be optimized further (Section 4.1 shows the benefit of optimization) or use a more uniform style of I/O. Thus, evaluation must focus on inherent patterns rather than making a direct quantitative comparison.

3 Observed Energy of Communication, Computation, and Compression

In this section, we observe that over 1000 32 bit ADD instructions can be executed by the Skiff with the same amount of energy it requires to send a single bit via wireless ethernet. This fact motivates the investigation of pre-transmission compression of data to reduce overall energy. Initial experiments reveal that reducing the number of bits to send does not always reduce the total energy of the task. This section elaborates on both of these points which necessitate the in-depth experiments of Section 3.3.

3.1 Raw Communication-to-Computation Energy Ratio

To quantify the gap between wireless communication and computation, we have measured wireless idle, send, and receive energies on the Skiff platform. To eliminate competition for wireless bandwidth from other devices in the lab, we established a dedicated channel and ran the network in ad-hoc mode consisting of only two wireless nodes. We streamed UDP packets from one node to the other; UDP was used to eliminate the effects of waiting for an ACK. This also insures that receive tests measure only receive energy and send tests measure only send energy. This setup is intended to find the minimum network energy by removing arbitration delay and the energy of TCP overhead to avoid biasing our results.

With the measured energy of the transmission and the size of data file, the energy required to send or receive a bit can be derived. The results of these network benchmarks appear in Figure 3 and are consistent with other studies [20]. The card is set to its maximum speed of

11 Mb/s and two tests are conducted. In the first, the Skiff communicates with a wireless card mere inches away and achieves 5.70 Mb/sec. In the second, the second node is placed as far from the Skiff as possible without losing packets. Only 2.85 Mb/sec is achieved. These two cases bound the performance of our 11 Mb/sec wireless card; typical performance should be somewhere between them.

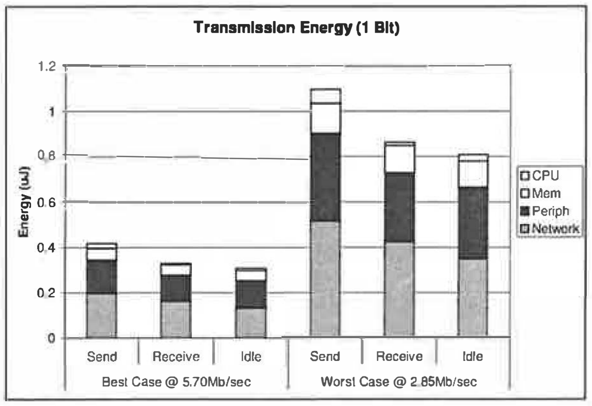


Figure 3. Measured communication energy of Enterasys wireless NIC

Next, a microbenchmark is used to determine the minimum energy for an ADD instruction. We use Linux boot code to bootstrap the processor; select a cache configuration; and launch assembly code unencumbered by an operating system. One thousand ADD instructions are followed by an unconditional branch which repeats them. This code was chosen and written in assembly language to minimize effects of the branch. Once the program has been loaded into instruction cache, the energy used by the processor for a single add is 0.86 nJ.

From these initial network and ADD measurements, we can conclude that sending a single bit is roughly equivalent to performing 485–1267 ADD operations depending on the quality of the network link ($\frac{4.17\times10^{-7}\,J}{0.86\times10^{-9}\,J} \approx 485$ or $\frac{1.09\times10^{-6}\,J}{0.86\times10^{-9}\,J} \approx 1267$). This gap of 2–3 orders of magnitude suggests that much additional effort can be spent trying to reduce a file's size before it is sent or received. But the issue is not so simple.

3.2 Application-Level Communication-to-Computation Energy Ratio

On the Skiff platform, memory, peripherals, and the network card remain powered on even when they are not active, consuming a fixed energy overhead. They may even switch when not in use in response to changes on shared buses. The energy used by these components during the ADD loop is significant and is shown in Table 2. Once a task-switching operating system is loaded and other applications vie for processing time, the communication-to-computation energy ratio will decrease further. Finally, the applications examined in this paper are more than a mere series of ADDs; the variety of instructions (especially Loads and Stores) in compression applications shrinks the ratio further.

Network card	0.43 nJ
CPU	**0.86 nJ**
Mem	1.10 nJ
Periph	4.20 nJ
Total	6.59 nJ

Table 2. Total Energy of an ADD

The first row of Figures 4 and 5 show the energy required to compress our text and web dataset and transmit it via wireless ethernet. To avoid punishing the benchmarks for the Skiff's high power, idle energy has been removed from the peripheral component so that it represents only the amount of *additional* energy (due to bus toggling and arbitration effects) over and above the energy that would have been consumed by the peripherals remaining idle for the duration of the application. Idle energy is not removed from the memory and CPU portions as they are required to be active for the duration of the application. The network is assumed to consume no power until it is turned on to send or receive data. The popular compression applications discussed in Section 2.2 are used with their default parameters, and the rightmost bar shows the energy of merely copying the uncompressed data over the network. Along with energy due to default operation (labeled "bzip2-900," "compress-16," "lzo-16," "ppmd-10240," and "zlib-6"), the figures include energy for several invocations of each application with varying parameters. *bzip2* is run with both the default 900 KB block sizes as well as its smallest 100 KB block. *compress* is also run at both ends of its spectrum (12 bit and 16 bit maximum codeword size). *LZO* runs in just 16 KB of working memory. *PPMd* uses 10 MB, 1 MB, and 32 KB memory with the cutoff mechanism for freeing space (as it is faster than the default "restart" in low-memory configurations). *zlib* is run in a configuration similar to *gzip*. The numeric suffix (9, 6, or 1) refers to effort level and is analogous to *gzip*'s commandline option. These various invocations will be studied in section 3.3.3.

While most compressors do well with the web data, in several cases the energy to compress the file approaches or outweighs the energy to transmit it. This problem is even worse for the harder-to-compress text data. The second row of Figures 4 and 5 shows the reverse operation: receiving data via wireless ethernet and decompressing it. The decompression operation is usually less costly

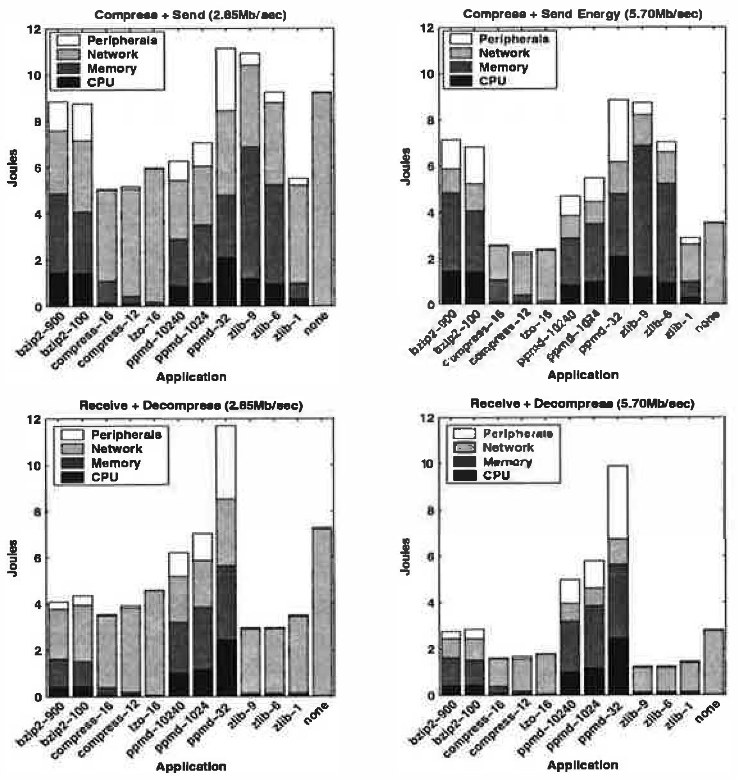


Figure 4. Energy required to transmit 1MB compressible text data

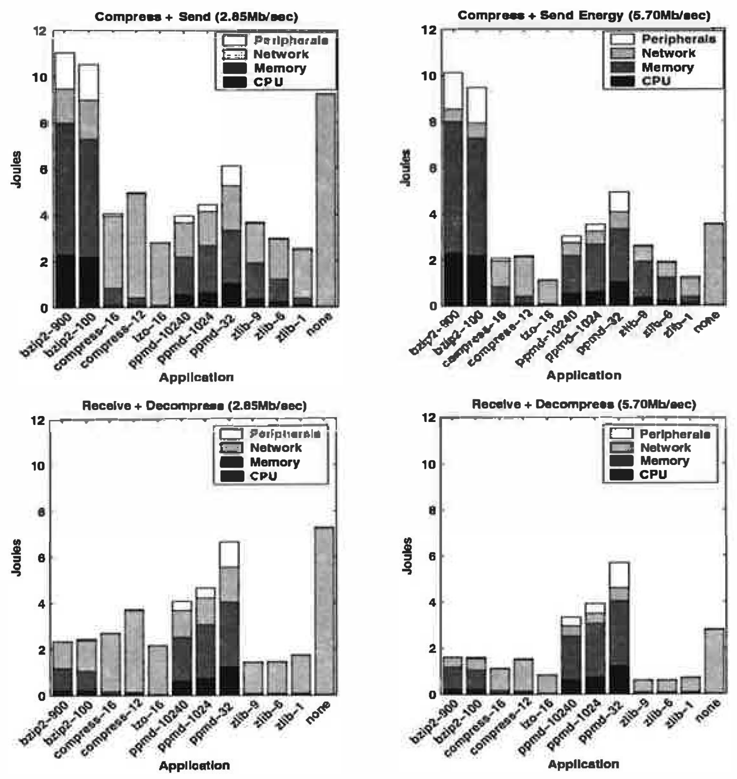


Figure 5. Energy required to transmit 1MB compressible web data

than compression in terms of energy, a fact which will be helpful in choosing a low-energy, asymmetric, lossless compression scheme. As an aside, we have seen that as transmission speed increases, the value of reducing wireless energy through data compression is less. Thus, even when compressing and sending data appears to require the same energy as sending uncompressed data, it is beneficial to apply compression for the greater good: more shared bandwidth will be available to all devices allowing them to send data faster and with less energy. Section 3.3 will discuss how such high net energy is possible despite the motivating observations.

3.3 Energy analysis of popular compressors

We will look deeper into the applications to discover why they cannot exploit the communication - computation energy gap. To perform this analysis, we rely on empirical observations on the Skiff platform as well as the execution-driven simulator known as SimpleScalar [7]. Though SimpleScalar is inherently an out-of-order, superscalar simulator, it has been modified to read statically linked ARM binaries and model the five-stage, in-order pipeline of the SA-110x [2]. As SimpleScalar is beta software we will handle the statistics it reports with caution, using them to explain the *traits* of the compression applications rather than to describe their precise execution on a Skiff. Namely, high instruction counts and high cost of memory access lead to poor energy efficiency.

3.3.1 Instruction count

We begin by looking at the number of instructions each requires to remove and restore a bit (Table 3). The range of instruction counts is one empirical indication of the applications' varying complexity. The excellent performance of *LZO* is due in part to its implementation as a single function, thus there is no function call overhead. In addition, *LZO* avoids superfluous copying due to buffering (in contrast with *compress* and *zlib*). As we will see, the number of memory accesses plays a large role in determining the speed and energy of an application. Each program contains roughly the same percentage of loads and stores, but the great difference in dynamic number of instructions means that programs such as *bzip2* and *PPMd* (each executing over 1 billion instructions) execute more total instructions and therefore have the most memory traffic.

3.3.2 Memory hierarchy

One noticeable similarity of the bars in Figures 4 and 5 is that the memory requires more energy than the processor. To pinpoint the reason for this, microbenchmarks were run on the Skiff memory system.

The SA-110 data cache is 16 KB. It has 32-way associativity and 16 sets. Each block is 32 bytes. Data is evicted at half-block granularity and moves to a 16 entry-by-16 byte write buffer. The write buffer also collects stores that miss in the cache (the cache is writeback/non-write-allocate). The store buffer can merge stores to the same entry.

The hit benchmark accesses the same location in memory in an infinite loop. The miss benchmark consecutively accesses the entire cache with a 32 byte stride followed by the same access pattern offset by 16 KB. Writebacks are measured with a similar pattern, but each load is followed by a store to the same location that dirties the block forcing a writeback the next time that location is read. Store hit energy is subtracted from the writeback energy. The output of the compiler is examined to insure the correct number of load or store instructions is generated. Address generation instructions are ignored for miss benchmarks as their energy is minimal compared to that of a memory access. When measuring store misses in this fashion (with a 32 byte stride), the worse-case behavior of the SA-110's store buffer is exposed as no writes can be combined. In the best case, misses to the the same buffered region can have energy similar to a store hit, but in practice, the majority of store misses for the compression applications are unable to take advantage of batching writes in the store buffer.

Table 4 shows that hitting in the cache requires more energy than an ADD (Table 2), and a cache miss requires up to 145 times the energy of an ADD. Store misses are less expensive as the SA-110 has a store buffer to batch accesses to memory. To minimize energy, then, we must seek to minimize cache-misses which require prolonged access to higher voltage components.

3.3.3 Minimizing memory access energy

One way to minimize misses is to reduce the memory requirements of the application. Figure 6 shows the effect of varying memory size on compression/decompression time and compression ratio. Looking back at Figures 4 and 5, we see the energy implications of choosing the right amount of memory. Most importantly, we see that merely choosing the fastest or best-compressing application does not result in lowest overall energy. Table 5 notes the throughput of each application; we see that with the Skiff's processor, several applications have difficulty meeting the line rate of the network which may preclude their use in latency-critical applications.

In the case of *compress* and *bzip2*, a larger memory footprint stores more information about the data and can be used to improve compression ratio. However, storing more information means less of the data fits in the cache leading to more misses, longer runtime and hence more

	bzip2	compress	LZO	PPMd	zlib
Compress: instructions per bit removed (Text Data)	116	10	7	76	74
Decompress: instructions per bit restored (Text Data)	31	6	2	10	5
Compress: instructions per bit removed (Web Data)	284	9	2	60	23
Decompress: instructions per bit restored (Web Data)	20	5	1	79	3

Table 3. Instructions per bit

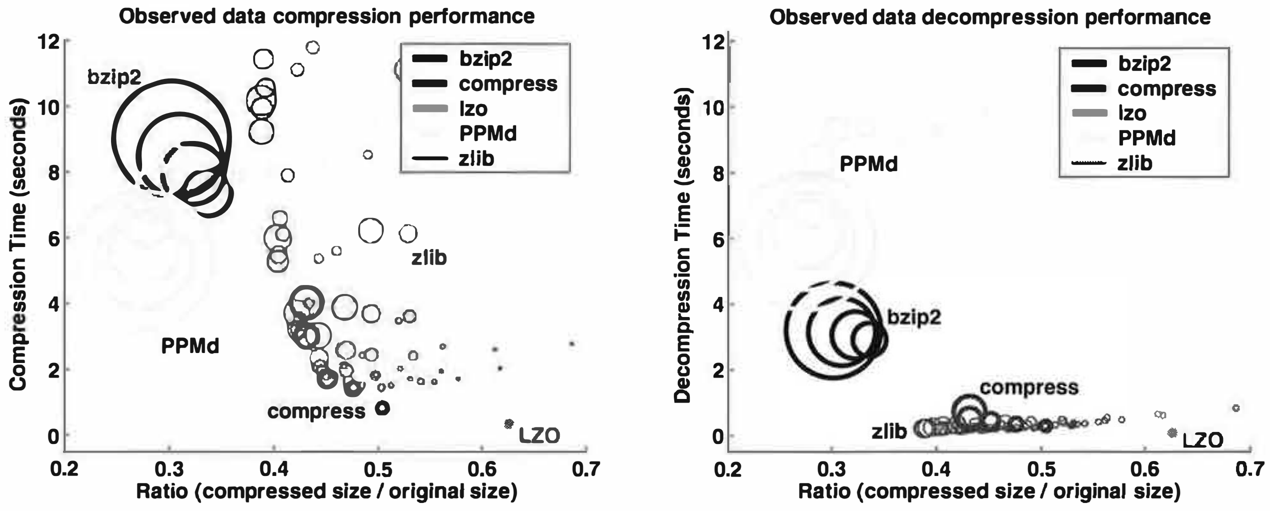


Figure 6. Memory, time, and ratio (Text data). Memory footprint is indicated by area of circle; footprints shown range from 3KB - 8MB

	Cycles	Energy (nJ)
Load Hit	1	2.72
Load Miss	80	124.89
Writeback	107	180.53
Store Hit	1	2.41
Store Miss	33	78.34
ADD	1	0.86

Table 4. Measured memory energy vs. ADD energy

energy. This tradeoff need not apply in the case where more memory allows a more efficient data structure or algorithm. For example, *bzip2* uses a large amount of memory, but for good reason. While we were able to implement its sort with the quicksort routine from the standard C library to save significant memory, the compression takes over 2.5 times as long due to large constants in the runtime of the more traditional quicksort in the standard library. This slowdown occurs even when 16KB block sizes [38] are used to further reduce memory requirements. Once *PPMd* has enough memory to do useful work, more context information can be stored and less complicated escape handling is necessary.

The widely scattered performance of *zlib*, even with similar footprints, suggest that one must be careful in choosing parameters for this library to achieve the desired goal (speed or compression ratio). Increasing window size effects compression; for a given window, a larger hash table improves speed. Thus, the net effect of more memory is variable. The choice is especially important if memory is constrained as certain window/memory combinations are inefficient for a particular speed or ratio.

The decompression side of the figure underscores the valuable asymmetry of some of the applications. Often decompressing data is a simpler operation than compression which requires less memory (as in *bzip2* and *zlib*). The simple task requires a relatively constant amount of time as there is less work to do: no sorting for *bzip2* and no searching though a history buffer for *zlib*, *LZO*, and *compress* because all the information to decompress a file is explicit. The contrast between compression and decompression for *zlib* is especially large. PPM implementations must go through the same procedure to decompress a file, undoing the arithmetic coding and building a model to keep its probability counts in sync with the compressor's. The arithmetic coder/decoder used in *PPMd* requires more time to decode than encode, so decompression requires more time.

Each of the applications examined allocates fixed-size

	bzip2	compress	LZO	PPMd	zlib
Compress read throughput (Text data)	0.91	3.70	24.22	1.57	0.82
Decompress write throughput (Text data)	2.59	11.65	109.44	1.42	41.15
Compress read throughput (Web data)	0.58	4.15	50.05	2.00	3.29
Decompress write throughput (Web data)	3.25	27.43	150.70	1.75	61.29

Table 5. Application throughputs (Mb/sec)

structures regardless of the input data length. Thus, in several cases more memory is set aside than is actually required. However, a large memory footprint may not be detrimental to an application if its current working set fits in the cache. The simulator was used to gather cache statistics. PPM and BWT are known to be quite memory intensive. Indeed, *PPMd* and *bzip2* access the data cache 1–2 orders of magnitude more often than the other benchmarks. *zlib* accesses data cache almost as much as *PPMd* and *bzip2* during compression, but drops from 150 million accesses to 8.2 million during decompression. Though LZ77 is local by nature, the large window and data structures hurt its cache performance for *zlib* during the compression phase. *LZO* also uses LZ77, but is designed to require just 16KB of memory and goes to main memory over five times less often than the next fastest application. The followup to the SA-110 (the SA-1110 used in Compaq's iPAQ handheld computer) has only an 8KB data cache which would exaggerate any penalties observed here. Though large, low-power caches are becoming possible (the X-Scale has two 32KB caches), as long as the energy of going to main memory remains so much higher, we must be concerned with cache misses.

3.4 Summary

On the Skiff, compression and decompression energy are roughly proportional to execution time. We have seen that the Skiff requires lots of energy to work with aggressively compressed data due to the amount of high-latency/high-power memory references. However using the fastest-running compressor or decompressor is not necessarily the best choice to minimize *total* transmission energy. For example, during decompression both *zlib* and *compress* run slower than *LZO*, but they receive fewer bits due to better compression so total energy is less than *LZO*. These applications successfully walk the tightrope of computation versus communication cost. Despite the greater energy needed to decompress the data, the decrease in receive energy makes the net operation a win. More importantly, we have shown that reducing energy is not as simple as choosing the fastest or best-compressing program.

We can generalize the results obtained on the Skiff in the following fashion. Memory energy is some multiple of CPU energy. Network energy (send and receive) is a far greater multiple of CPU energy. It is difficult to predict how quickly energy of components will change over time. Even predicting whether a certain component's energy usage will grow or shrink can be difficult. Many researchers envision ad-hoc networks made of nearby nodes. Such a topology, in which only short-distance wireless communication is necessary, could reduce the energy of the network interface relative to the CPU and memory. On the other hand, for a given mobile CPU design, planned manufacturing improvements may lower its relative power and energy. Processors once used only in desktop computers are being recast as mobile processors. Though their power may be much larger than that of the Skiff's StrongARM, higher clock speeds may reduce energy. If one subscribes to the belief that CPU energy will steadily decrease while memory and network energy remain constant, then *bzip2* and *PPMd* become viable compressors. If both memory and CPU energy decrease, then current low-energy compression tools (*compress* and *LZO*) can even be surpassed by their computation and memory intensive peers. However, if only network energy decreases while the CPU and memory systems remain static, energy-conscious systems may forego compression altogether as it now requires more energy than transmitting raw data. Thus, it is important for software developers to be aware of such hardware effects if they wish to keep compression energy as low as possible. Awareness of the type of data to be transmitted is important as well. For example, transmitting our world-wide-web data required less energy in general than the text data. Trying to compress pre-compressed data (not shown) requires significantly more energy and is usually futile.

4 Results

We have seen energy can be saved by compressing files before transmitting them over the network, but one must be mindful of the energy required to do so. Compression and decompression energy may be minimized through wise use of memory (including efficient data structures and/or sacrificing compression ratio for cacheability). One must be aware of evolving hardware's effect on overall energy. Finally, knowledge of compression and decompression energy for a given system permits the use of asymmetric compression in which the lowest energy application for compression is paired with the lowest energy application for decompression.

4.1 Understanding cache behavior

Figure 7 shows the compression energy of several successive optimizations of the *compress* program. The baseline implementation is itself an optimization of the original *compress* code. The number preceding the dash refers to the maximum length of codewords. The graph illustrates the need to be aware of the cache behavior of an application in order to minimize energy. The data structure of *compress* consists of two arrays: a hash table to store symbols and prefixes, and a code table to associate codes with hash table indexes. The tables are initially stored back-to-back in memory. When a new symbol is read from the input, a single index is used to retrieve corresponding entries from each array. The "16-merge" version combines the two tables to form an array of structs. Thus, the entry from the code table is brought into the cache when the hash entry is read. The reduction in energy is negligible: though one type of miss has been eliminated, the program is actually dominated by a second type of miss: the probing of the hash table for free entries. The Skiff data cache is small (16KB) compared to the size of the hash table (≈270KB), thus the random indexing into the hash table results in a large number of misses. A more useful energy and performance optimization is to make the hash table more sparse. This admits fewer collisions which results in fewer probes and thus a smaller number of cache misses. As long as the extra memory is available to enable this optimization, about 0.53 Joules are saved compared with applying no compression at all. This is shown by the "16-sparse" bar in the figure. The baseline and "16-merge" implementations require more energy than sending uncompressed data. A 12-bit version of compress is shown as well. Even when peripheral overhead energy is disregarded, it outperforms or ties the 16-bit schemes as its reduced memory energy due to fewer misses makes up for poorer compression.

Another way to reduce cache misses is to fit both tables completely in the cache. Compare the following two structures:

```
struct entry{                struct entry{
  int fcode;                   signed fcode:20;
  unsigned short code;         unsigned code:12;
}table[SIZE];                }table[SIZE];
```

Each `entry` stores the same information, but the array on the left wastes four bytes per entry. Two bytes are used only to align the short `code`, and overly-wide

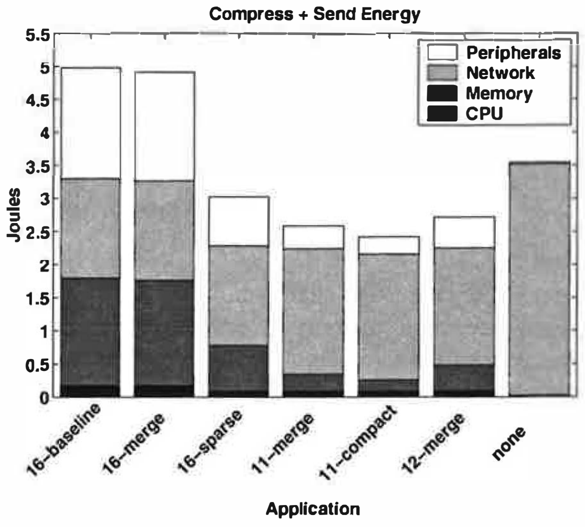


Figure 7. Optimizing *compress* (Text data)

types result in twelve wasted bits in `fcode` and four bits wasted in `code`. Using bitfields, the layout on the right contains the same information yet fits in half the space. If the entry were not four bytes, it would need to contain more members for alignment. Code with such structures would become more complex as C does not support arrays of bitfields, but unless the additional code introduces significant instruction cache misses, the change is low-impact. A bitwise AND and a shift are all that is needed to determine the offset into the compact structure. By allowing the whole table to fit in the cache, the program with the compacted array has just 56,985 data cache misses compared with 734,195 in the un-packed structure; a 0.0026% miss rate versus 0.0288%. The energy benefit for *compress* with the compact layout is negligible because there is so little CPU and memory energy to eliminate by this technique. The "11-merge" and "11-compact" bars illustrate the similarity. Nevertheless, 11-compact runs 1.5 times faster due to the reduction in cache misses, and such a strategy could be applied to any program which needs to reduce cache misses for performance and/or energy. Eleven bit codes are necessary even with the compact layout in order to reduce the size of the data structure. Despite a dictionary with half the size, the number of bytes to transmit increases by just 18% compared to "12-merge." Energy, however, is lower with the smaller dictionary due to less energy spent in memory and increased speeds which reduce peripheral overhead.

4.2 Exploiting the sleep mode

It has been noted that when a platform has a low-power idle state, it may be sensible to sacrifice energy in the short-term in order to complete an application quickly and enter the low-power idle state [26]. Figure 8 shows the effect of this analysis for compression and sending of text. Receive/decompression exhibits similar, but less-pronounced variation for different idle powers. It is interesting to note that, assuming a low-power idle mode can be entered once compression is complete, one's choice of compression strategies will vary. With its 1 Watt of idle power, the Skiff would benefit most from *zlib* compression. A device which used negligible power when idle would choose the *LZO* compressor. While *LZO* does not compress data the most, it allows the system to drop into low-power mode as quickly as possible, using less energy when long idle times exist. For web data (not shown due to space constraints) the compression choice is *LZO* when idle power is low. When idle power is one Watt, *bzip2* energy is over 25% more energy efficient than the next best compressor.

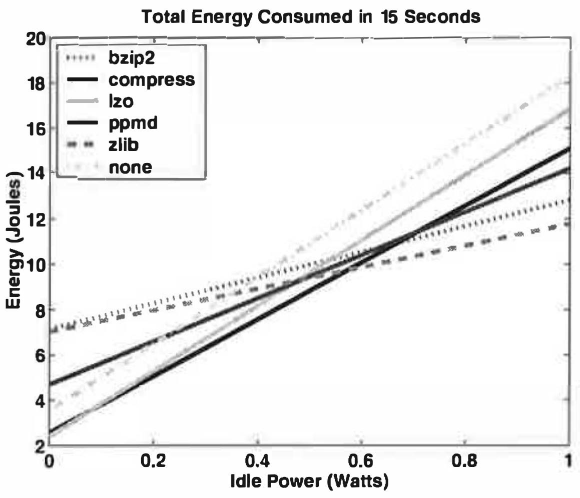


Figure 8. Compression + Send energy consumption with varying sleep power (Text data)

4.3 Asymmetric compression

Consider a wireless client similar to the Skiff exchanging English text with a server. All requests by the client should be made with its minimal-energy compressor, and all responses by the server should be compressed in such a way that they require minimal decompression energy at the client. Recalling Figures 4 and 5, and recognizing that the Skiff has no low-power sleep mode, we choose "compress-12" (the twelve-bit codeword LZW compressor) for our text compressor as it provides the lowest total compression energy over all communication speeds.

To reduce decompression energy, the client can request data from the server in a format which facilitates low-energy decompression. If latency is not critical and the client has a low-power sleep mode, it can even wait while the server converts data from one compressed format to another. On the Skiff, *zlib* is the lowest energy decompressor for both text and web data. It exhibits the property that regardless of the effort and memory parameters used to compress data, the resulting file is quite easy to decompress. The decompression energy difference between *compress*, *LZO*, and *zlib* is minor at 5.70 Mb/sec, but more noticeable at slower speeds.

Figure 9 shows several other combinations of compressor and decompressor at 5.70 Mb/sec. "zlib-9 + zlib-9" represents the symmetric pair with the least decompression energy, but its high compression energy makes it unlikely to be used as a compressor for devices which must limit energy usage. "compress-12 + compress-12" represents the symmetric pair with the least compression energy. If symmetric compression and decompression is desired, then this "old-fashioned" Unix compress program can be quite valuable. Choosing "zlib-1" at both ends makes sense as well – especially for programs linked with the zlib library. Compared with the minimum symmetric compressor-decompressor, asymmetric compression on the Skiff saves only 11% of energy. However, modern applications such as ssh and mod_gzip use "zlib-6" at both ends of the connection. Compared to this common scheme, the optimal asymmetric pair yields a 57% energy savings – mostly while performing compression.

It is more difficult to realize a savings over symmetric zlib-6 for web data as all compressors do a good job compressing it and "zlib-6" is already quite fast. Nevertheless, by pairing "lzo" and "zlib-9," we save 12% of energy over symmetric "lzo" and 31% over symmetric "zlib-6."

5 Related work

This section discusses data compression for low-bandwidth devices and optimizing algorithms for low energy. Though much work has gone into these fields individually, it is difficult to find any which combines them to examine lossless data compression from an energy standpoint. Computation-to-communication energy ratio has been been examined before [12], but this work adds physical energy measurements and applies the results to lossless data compression.

5.1 Lossless Data compression for low-bandwidth devices

Like any optimization, compression can be applied at many points in the hardware-software spectrum. When

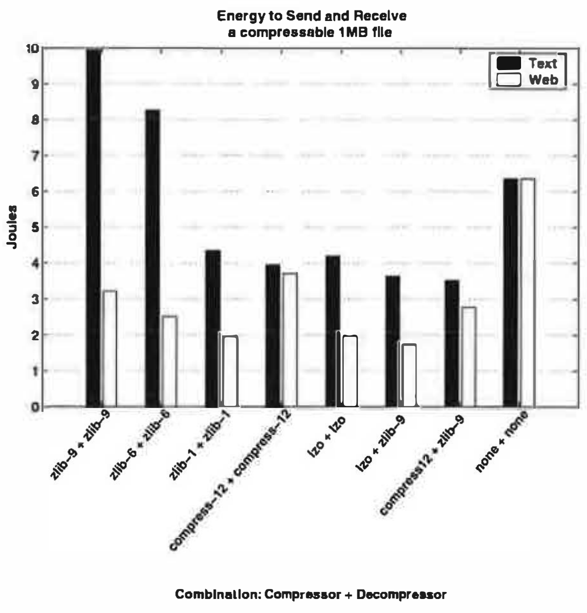


Figure 9. Choosing an optimal compressor-decompressor pair

applied in hardware, the benefits and costs propagate to all aspects of the system. Compression in software may have a more dramatic effect, but for better or worse, its effects will be less global.

The introduction of low-power, portable, low-bandwidth devices has brought about new (or rediscovered) uses for data compression. Van Jacobson introduced TCP/IP Header Compression in RFCl 144 to improve interactive performance over low-speed (wired) serial links [19], but it is equally applicable to wireless. By taking advantage of uniform header structure and self-similarity over the course of a particular networked conversation, 40 byte headers can be compressed to 3–5 bytes. Three byte headers are the common case. An all-purpose header compression scheme (not confined to TCP/IP or any particular protocol) appears in [24]. TCP/IP payloads can be compressed as well with IP-Comp [39], but this can be wasted effort if data has already been compressed at the application layer.

The Low-Bandwidth File System (LBFS) exploits similarities between the data stored on a client and server, to exchange only data blocks which differ [31]. Files are divided into blocks with content-based fingerprint hashes. Blocks can match any file in the file system or the client cache; if client and server have matching block hashes, the data itself need not be transmitted. Compression is applied before the data is transmitted. Rsync [44] is a protocol for efficient file transfer which preceded LBFS. Rather than content-based fingerprints, Rsync uses its rolling hash function to account for changes in block size. Block hashes are compared for a pair of files to quickly identify similarities between client and server. Rsync block sharing is limited to files of the same name.

A protocol-independent scheme for text compression, NCTCSys, is presented in [30]. NCTCSys involves a common dictionary shared between client and server. The scheme chooses the best compression method it has available (or none at all) for a dataset based on parameters such as file size, line speed, and available bandwidth.

Along with remote proxy servers which may cache or reformat data for mobile clients, splitting the proxy between client and server has been proposed to implement certain types of network traffic reduction for HTTP transactions [14, 23]. Because the delay required for manipulating data can be small in comparison with the latency of the wireless link, bandwidth can be saved with little effect on user experience. Alternatively, compression can be built into servers and clients as in the *mod_gzip* module available for the Apache webserver and HTTP 1.1 compliant browsers [16]. Delta encoding, the transmission of only parts of documents which differ between client and server, can also be used to compress network traffic [15, 27, 28, 35].

5.2 Optimizing algorithms for low energy

Advanced RISC Machines (ARM) provides an application note which explains how to write C code in a manner best-suited for its processors and ISA [1]. Suggestions include rewriting code to avoid software emulation and working with 32 bit quantities whenever possible to avoid a sign-extension penalty incurred when manipulating shorter quantities. To reduce energy consumption and improve performance, the OptAlg tool represents polynomials in a manner most efficient for a given architecture [34]. As an example, cosine may be expressed using two MAC instructions and an MUL to apply a "Horner transform" on a Taylor Series rather than making three calls to a cosine library function.

Besides architectural constraints, high level languages such as C may introduce false dependencies which can be removed by disciplined programmers. For instance, the use of a global variable implies loads and stores which can often be eliminated through the use of register-allocated local variables. Both types of optimizations are used as guidelines by PHiPAC [6], an automated generator of optimized libraries. In addition to these general coding rules, architectural parameters are provided to a code generator by search scripts which work to find the best-performing routine for a given platform.

Yang et al. measured the power and energy impact of various compiler optimizations, and reached the conclusion that energy can be saved if the compiler can reduce execution time and memory references [48]. Šimunić found that floating point emulation requires much energy due to the sheer number of extra instructions required [46]. It was also discovered that instruction flow optimizations (such as loop merging, unrolling, and software pipelining) and ISA specific optimizations (e.g., the use of a multiply-accumulate instruction) were not applied by the ARM compiler and had to be introduced manually. Writing such energy-efficient source code saves more energy than traditional compiler speed optimizations [45].

The CMU Odyssey project studied "application-aware adaptation" to deal with the varying, often limited resources available to mobile clients. Odyssey trades data quality for resource consumption as directed by the operating system. By placing the operating system in charge, Odyssey balances the needs of all running applications and makes the choice best suited for the system. Application-specific adaptation continues to improve. When working with a variation of the Discrete Cosine Transform and computing first with DC and low-frequency components, an image may be rendered at 90% quality using just 25% of its energy budget [41]. Similar results are shown for FIR filters and beamforming using a most-significant-first transform. Parameters used by JPEG lossy image compression can be varied to reduce bandwidth requirements and energy consumption for particular image quality requirements [43]. Research to date has focused on situations where energy-fidelity tradeoffs are available. Lossless compression does not present this luxury because the original bits must be communicated in their entirety and re-assembled in order at the receiver.

6 Conclusion and Future Work

The value of this research is not merely to show that one can optimize a given algorithm to achieve a certain reduction in energy, but to show that the choice of how and whether to compress is not obvious. It is dependent on hardware factors such as relative energy of CPU, memory, and network, as well as software factors including compression ratio and memory access patterns. These factors can change, so techniques for lossless compression prior to transmission/reception of data must be re-evaluated with each new generation of hardware and software. On our StrongARM computing platform, measuring these factors allows an energy savings of up to 57% compared with a popular default compressor and decompressor. Compression and decompression often have different energy requirements. When one's usage supports the use of asymmetric compression and decompression, up to 12% of energy can be saved compared with a system using a single optimized application for both compression and decompression.

When looking at an entire system of wireless devices, it may be reasonable to allow some to individually use more energy in order to minimize the total energy used by the collection. Designing a low-overhead method for devices to cooperate in this manner would be a worthwhile endeavor. To facilitate such dynamic energy adjustment, we are working on EProf: a portable, realtime, energy profiler which plugs into the PC-Card socket of a portable device [22]. EProf could be used to create feedback-driven compression software which dynamically tunes its parameters or choice of algorithms based on the measured energy of a system.

7 Acknowledgements

Thanks to John Ankcorn, Christopher Batten, Jamey Hicks, Ronny Krashinsky, and the anonymous reviewers for their comments and assistance. This work is supported by MIT Project Oxygen, DARPA PAC/C award F30602-00-2-0562, NSF CAREER award CCR-0093354, and an equipment grant from Intel.

References

[1] Advanced RISC Machines Ltd (ARM). *Writing Efficient C for ARM*, Jan. 1998. Application Note 34.

[2] T. M. Austin and D. C. Burger. SimpleScalar version 4.0 release. *Tutorial in conjunction with 34th Annual International Symposium on Microarchitecture*, Dec. 2001.

[3] T. Bell and D. Kulp. Longest match string searching for Ziv-Lempel compression. Technical Report 06/89, Department of Computer Science, University of Canterbury, New Zealand, 1989.

[4] T. Bell, M. Powell, J. Horlor, and R. Arnold. The Canterbury Corpus. http://www.corpus.canterbury.ac.nz/.

[5] T. Bell, I. H. Witten, and J. G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–591, 1989.

[6] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *11th ACM International Conference on Supercomputing*, July 1997.

[7] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[8] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Systems Research Center, May 1994.

[9] J. Gailly and M. Adler. zlib. http://www.gzip.org/zlib.

[10] J. Gailly, Maintainer. `comp.compression` Internet newsgroup: Frequently Asked Questions, Sept. 1999.

[11] J. Gilchrist. Archive comparison test. http://compression.ca.

[12] P. J. Havinga. Energy efficiency of error correction on wireless systems. In *IEEE Wireless Communications and Networking Conference*, Sept. 1999.

[13] J. Hicks et al. Compaq personal server project, 1999. http://crl.research.compaq.com /projects/personalserver/default.htm.

[14] B. C. Housel and D. B. Lindquist. Webexpress: a system for optimizing web browsing in a wireless environment. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, 1996.

[15] J. J. Hunt, K.-P. Vo, and W. F. Tichy. An empirical study of delta algorithms. In *Software configuration management: ICSE 96 SCM-6 Workshop*. Springer, 1996.

[16] Hyperspace Communications, Inc. Mod_gzip. http://www.ehyperspace.com /htmlonly/products/mod_gzip.html.

[17] Intel Corporation. *SA-110 Microprocessor Technical Reference Manual*, December 2000.

[18] Intel Corporation. *Intel StrongARM SA-1110 Microprocessor Developer's Manual*, October 2001.

[19] V. Jacobson. RFC 1144: Compressing TCP/IP headers for low-speed serial links, Feb. 1990.

[20] K. Jamieson. Implementation of a power-saving protocol for ad hoc wireless networks. Master's thesis, Massachusetts Institute of Technology, Feb. 2002.

[21] P. Jannesen et. al. (n)compress. available, among other places, in Redhat 7.2 distribution of Linux.

[22] K. Koskelin, K. Barr, and K. Asanović. Eprof: An energy profiler for the iPaq. In *2nd Annual Student Oxygen Workshop*. MIT Project Oxygen, 2002.

[23] R. Krashinsky. Efficient web browsing for mobile clients using HTTP compression. Technical Report MIT-LCS-TR-882, MIT Lab for Computer Science, Jan. 2003.

[24] J. Lilley, J. Yang, H. Balakrishnan, and S. Seshan. A unified header compression framework for low-bandwidth links. In *6th ACM MOBICOM*, Aug. 2000.

[25] Lycos. Lycos 50, Sept. 2002. Top 50 searches on Lycos for the week ending September 21, 2002.

[26] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *International Conference on Supercomputing*, June 2002.

[27] J. C. Mogul. Trace-based analysis of duplicate suppression in HTTP. Technical Report 99.2, Compaq Computer Corporation, Nov. 1999.

[28] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. Technical Report 97/4a, Compaq Computer Corporation, Dec. 1997.

[29] J. Montanaro et al. A 160-mhz, 32-b, 0.5-w CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11), Nov. 1996.

[30] N. Motgi and A. Mukherjee. Network conscious text compression systems (NCTCSys). In *Proceedings of International Conference on Information and Theory: Coding and Computing*, 2001.

[31] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, Chateau Lake Louise, Banff, Canada, October 2001.

[32] Nielsen NetRatings Audience Measurement Service. Top 25 U.S Properties; Week of Sept 15th., Sept. 2002.

[33] M. F. Oberhumer. LZO. http://www.oberhumer.com/opensource/lzo/.

[34] A. Peymandoust, T. Šimunić, and G. D. Micheli. Low power embedded software optimization using symbolic algebra. In *Design, Automation and Test in Europe*, 2002.

[35] J. Santos and D. Wetherall. Increasing effective link bandwidth by suppressing replicated data. In *USENIX Annual Technical Conference*, June 1998.

[36] K. Sayood. *Introduction to data compression.* Morgan Kaufman Publishers, second edition, 2002.

[37] J. Seward. bzip2. http://www.spec.org /osg/cpu2000/CINT2000/256.bzip2/docs/256.bzip2.html.

[38] J. Seward. e2comp bzip2 library. http://cvs.bofh.asn.au/e2compr/index.html.

[39] A. Shacham, B. Monsour, R. Pereira, and M. Thomas. RFC 3173: IP payload compression protocol, Sept. 2001.

[40] D. Shkarin. PPMd. ftp://ftp.elf.stuba.sk/pub/pc/pack/ppmdi1.rar.

[41] A. Sinha, A. Wang, and A. Chandrakasan. Algorithmic transforms for efficient energy scalable computation. In *IEEE International Symposium on Low Power Electronics and Design*, August 2000.

[42] Standard Performance Evaluation Corporation. CPU2000, 2000.

[43] C. N. Taylor and S. Dey. Adaptive image compression for wireless multimedia communication. In *IEEE International Conference on Communication*, June 2001.

[44] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization.* PhD thesis, Australian National University, Apr. 2000.

[45] T. Šimunić, L. Benini, and G. D. Micheli. Energy-efficient design of battery-powered embedded systems. In *IEEE International Symposium on Low Power Electronics and Design*, 1999.

[46] T. Šimunić, L. Benini, G. D. Micheli, and M. Hans. Source code optimization and profiling of energy consumption in embedded systems. In *International Symposium on System Synthesis*, 2000.

[47] M. A. Viredaz and D. A. Wallach. Power evaluation of Itsy version 2.4. Technical Report TN-59, Compaq Computer Corporation, February 2001.

[48] H. Yang, G. R. Gao, A. Marquez, G. Cai, and Z. Hu. Power and energy impact of loop transformations. In *Workshop on Compilers and Operating Systems for Low Power 2001, Parallel Architecture and Compilation Techniques*, Sept. 2001.

Energy-Adaptive Display System Designs for Future Mobile Environments

Subu Iyer[†] Lu Luo[‡] Robert Mayo[†] Parthasarathy Ranganathan[†]

[†]*Hewlett Packard Labs*
1501 Page Mill Road MS 1177
Palo Alto California 94304 USA
{Subu.Iyer,Bob.Mayo}@hp.com
Partha.Ranganathan@hp.com

[‡]*School of Computer Science*
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213 USA
luluo@cs.cmu.edu

Abstract

The utility of a mobile computer, such as a laptop, is largely constrained by battery life. The display stands out as a major consumer of battery energy, so reducing that consumption is desirable. In this paper, we motivate and study **energy-adaptive display sub-systems** that match display energy consumption to the functionality required by the workload/user. Through a detailed characterization of display usage patterns, we show that screen usage of a typical user is primarily associated with content that could be displayed in smaller and simpler displays with significantly lower energy use. We propose example energy-adaptive designs that use emerging *OLED displays* and software optimizations that we call *dark windows*. Modeling the power benefits from this approach shows significant, though user-specific, energy benefits. Prototype implementations also show acceptability of the new user interfaces among users.

1 Introduction

With the increased acceptance and use of mobile devices such as laptops and pocket computers, mobile computing systems are rapidly becoming one of the key markets of interest for computing systems. Since the batteries on these mobile systems are typically limited in capacity, reducing the energy consumption is one of the key challenges in designing mobile systems.

Among the various components that contribute to the consumption of electrical energy, the display sub-system (the electronics associated with the visual representation of the data generated by the system - namely the display and the controller) often plays an important role. For example, Udani and Smith report that the display component of the system can easily take over half the total energy of a laptop system [7]. Similarly, Choi et al. report that the display component of the system consumed close to 61% of the total power of the system for a handheld device. Furthermore, unlike some of the other components of the system, display power consumption has traditionally been relatively invariant across technology shrinks and unamenable to opportunities to exploit "slack," [5] making it a likely greater fraction of the total power of future systems.

Previous approaches to reducing display power consumption have either focused on aggressively turning off the entire display when it is not being used or have resorted to designing systems with lower-quality or smaller sized displays to minimize power. However, new technologies, such as Organic Light Emitting Diodes (OLEDs), are becoming available that allow lower power consumption when a reduced area of the screen is in use.

We propose utilizing this new flexibility to reduce energy consumption. Our work is based on the intuition that different workloads and users have varying display needs. Having a *"one-size-fits-all"* display targeted at the needs of the most aggressive workload/user often leads to large energy inefficiencies in the display energy consumptions of other workloads and users. Consequently, an *energy-adaptive system design* that consumes energy only on portions and characteristics of the screen that are being used by the application and are relevant to the user, can achieve energy benefits.

In this work, we make two key contributions. First, we perform a detailed characterization of the display screen usage of a representative test user population. Our results indicate that, on average, our users use only about 60% of the screen area available to them. Additionally, screen usage is often associated with content that could have been equivalently displayed, with no loss in visual quality, on much simpler lower-power displays (lower size, resolution, color, brightness, refresh rates, etc.) Using a detailed analysis of the user traces, we also correlate our results to user and application behavior. To the best of our knowledge, our work is the first to identify, quantify, and analyze such mismatch opportunities in workload/user needs and current display properties. Overall, our results indicate that an energy-adaptive system design that matches display power consumption to the functionality required by the workload/user can significantly reduce the energy consumption of future display sub-systems.

Building on the insights from the above analysis, we propose example energy-adaptive display system designs. At the hardware level, our designs leverage emerging Organic LED (OLED) displays [3] that use energy proportional to the overall light output of the display. At the software level, we propose *dark windows* optimizations that enable the windowing environment to change the brightness and color of areas of the screen that are not of interest to the user. We model the power benefits and study the user experience with our designs. Our results indicate significant, though user-specific, energy reductions with acceptable user interfaces.

The rest of the paper is organized as follows. Section 2 discusses our user study in greater detail. Section 3 discusses the example energy-adaptive display systems that we consider and our experiences with those. Section 4 further explores the design space with energy-adaptive displays and Section 5 discusses related work. Section 6 concludes the paper and discusses future work.

2 User Study

This section presents the results from our user study characterizing typical system usage of a representative user test population. Our goal was to understand the screen usage patterns and identify opportunities for power reduction. Section 2.1 discusses our methodology and Section 2.2 discusses our results.

2.1 Methodology for the user study

Our user study is based on usage of the Microsoft Windows environment by seventeen users. We chose Microsoft Windows environments because of their widespread acceptance and representativeness of the general mobile market. The users were chosen to include characteristics representative of typical usage and cover a cross section of mobile system usage (administrative tasks, code development, personal productivity, entertainment, etc.). Since we were interested in understanding both current and future user behavior when using mobile environments, we studied both laptop and desktop users. (Many of our laptop users used their machines as their main machine - both as a desktop and a laptop.) The systems used by the test users include a variety of screen sizes and display resolutions. Column 2 in Figure 1 summarizes the properties of the systems used by our test population.

An application-level logger program was run on the users' machines for times ranging from 1 to 14 days. The logger program was used to collect periodic information about (i) the current window of focus – its size, its location, and its title and (ii) the size of total screen area used (all non-minimized windows). Our sampling rate was set to once a second. Screen savers were set to turn on after a reasonable time (1-5 minutes) to allow us to isolate only the usage patterns when the user was active.

Column 3 in Figure 1 summarizes the length of the user traces. The traces range from 9 hours to 346 hours. The variation in the traces represent the differences in how individual users used their machines during their participation in the study. Overall, our samples represent close to 100 days of continuous computer usage time.

Column 4 in Figure 1 summarizes the length of the "active" user traces, after factoring out the time spent in the screen saver as an indication of the time the user was idle. Traces are still collected during the time it takes for the screen saver to be activated, but given the length of our logs, the effect of this is minor. The sizes of our active user logs range from about 6 hours to 61 hours of computer usage per user. Given that this is the time we are interested in, the rest of the paper will focus on the active window usage without considering the time spent in the screen saver. Existing technologies can save power spent in the idle time by turning off the displays.

2.2 User study results

Average screen usage. Figure 1 summarizes the information about the screen usage. Columns 5 and 6 present the mean and standard deviation, per user, for the screen usage of the window of focus. For this study, we define

User	Display (column 2)	Log length (column 3)	Active samples (column 4)	Screen usage for active samples: Mean — Window of focus	Std. dev — Window of focus	Mean — Background windows	Std. dev — Background windows
Desktop user population							
1	19" 1024x768	210 hours	33 hours	62.8%	38.5%	10.6%	21.2%
2	21" 1280 x 1024	346 hours	61 hours	57.2%	22.3%	11.6%	28.5%
3	19" 1280 x 1024	214 hours	31 hours	46.3%	19.7%	30.4%	19.7%
4	19" 1280 x 1024	64 hours	43 hours	36.7%	14.5%	34.1%	8.8%
5	19" 1280 x 1024	253 hours	27 hours	44.5%	22.7%	32.6%	21.1%
6	21" 1280 x 1024	229 hours	31 hours	55.5%	18.4%	24.7%	17.8%
7	21" 1280 x 1024	235 hours	30 hours	57.5%	19.2%	20.0%	18.8%
8	17" 1024 x 768	135 hours	13 hours	85.2%	26.2%	9.7%	24.4%
Laptop user population							
9	13" 1280 x 1024	42 hours	23 hours	61.8%	21.6%	25.1%	22.3%
10	14" 1024 x 768	98 hours	54 hours	71.1%	25.4%	22.4%	23.9%
11	14" 1400 x 1050	57 hours	57 hours	37.4%	20.3%	7.2%	15.1%
12	14" 1024 x 768	20 hours	13 hours	93.7%	12.3%	2.3%	12.2%
13	15" 1024 x 768	169 hours	154 hours	43.3%	38.9%	17.5%	24.3%
14	13" 800 x 600	132 hours	6.2 hours	71.1%	37.6%	3.0%	15.0%
15	14"1024 x 768	9 hours	6.4 hours	44.1%	21.4%	10.3%	15.3%
16	14" 1400 x 1050	69 hours	15 hours	54.6%	25.9%	18.5%	17.5%
17	14" 1024x768	10 hours	6.0 hours	77.3%	36.8%	5.0%	17.0%
Average screen usage – window of focus: 58.8%; background windows: 16.7%							

Figure 1: *Key statistics from user study. Column 3 summarizes the length of the user traces while column 4 summarizes the length of the* active *user traces after factoring out the time the user was idle. The* window of focus *columns summarize the percentage of screen area used by the active window while the* background windows *columns summarize the percentage of area used by other non-minimized windows not hidden under the active window.*

the window of focus as the window that accepts keyboard or mouse input. In determining the size of the window of focus, we include the title bar and the scroll bar and other menu bars that are embedded in the window. Columns 7 and 8 present the mean and standard deviation for the *additional* screen area used by other non-minimized windows in the system (i.e., the area not hidden under the window of focus).

Focusing on the average screen usage for the window of focus from Figure 1, we can see that our test population uses anywhere from 37% to 94% of the total screen area available to them. An additional 2% to 34% of the screen is used by other background windows that are not active, yet are not minimized. The last row of Figure 1 indicates the average usage across our user population. This average is obtained by computing the arithmetic mean of the averages of the individual users. This ensures that the average is not biased by users with larger log lengths. On average, across all our users, typically only about 59% of the entire screen area is used by the window of focus, the primary area of interest to the user. An additional 17% of the screen, on average, is used for background windows that are not minimized. In both these cases, however, the standard deviations are fairly high indicating a wide range in the screen usage values associated with each sample.

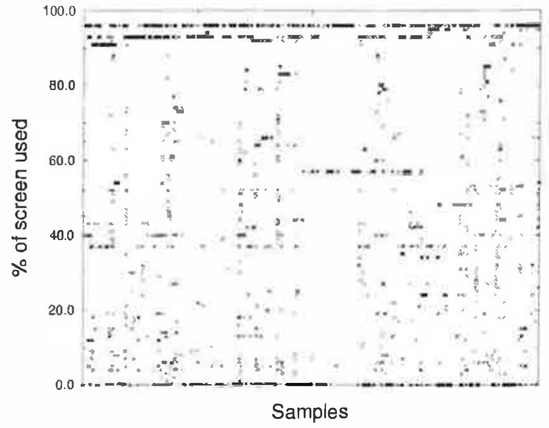


Figure 2: *Variation in the screen usage of the window of focus for typical user (User 1). Each point represents one data sample in the log.*

Screen usage distribution. To better understand the distribution of the screen usage characteristics, Figure 2 plots the variation in the screen usage of the window of focus for one sample user, over the log collection period.

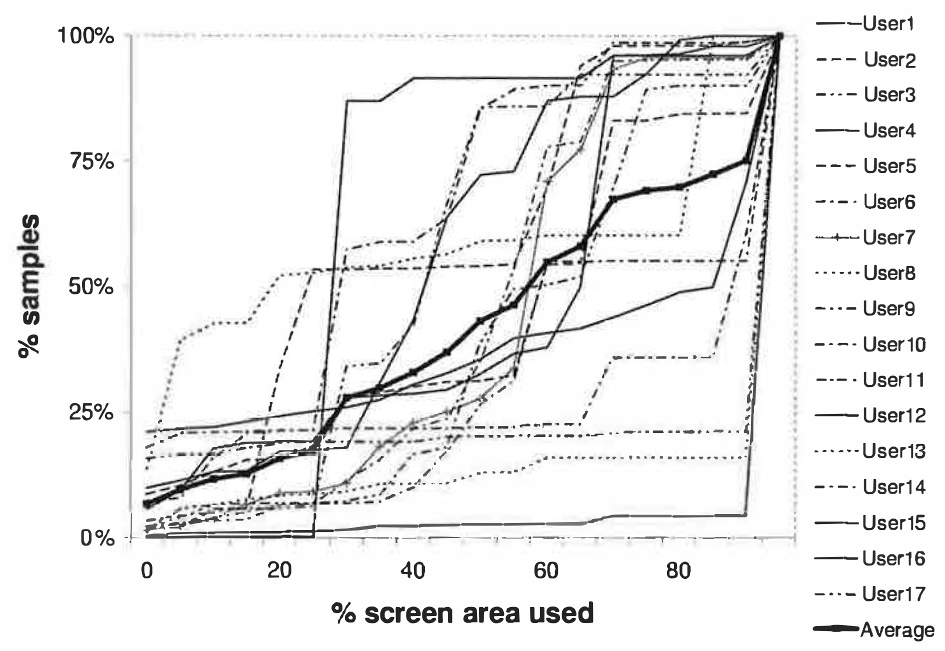


Figure 3: *Cumulative distributions of the active screen usages for the test population.*

Each point represents the average screen area associated with one data sample in the log. As can be seen from the figures, the percentage of screen usage varies significantly over the time the data was collected, all the way from near-0 to near-100% usage of the screen. Clustering of points at specific screen usage percentages can be correlated back to the continuous usage of key applications used by the user and their normal (or default) sizes.

Figure 3 presents the same data for all the users in a summarized manner. Each line in the graph represents one user from our test population and the thicker solid line represents values averaged over all the users. The X-axis represents the percentage of screen area used per sample and is divided into bins of 5 each. The Y axis represents the cumulative number of samples associated with each screen-area-percentage bin. For example, if we were to draw a vertical line from the 50% screen area point to intersect all the lines, that would give us the cumulative number of samples where each user uses less than 50% of the total available screen area. For example, focusing on *User 5*, this would mean that close to 54% of the samples use less than 50% of the screen area.

Summarizing the results in the graph, we can observe that, on average, across all our users, for almost 45% of the time, we end up using less than half the entire screen area. Some users spend more time in windows less than half the screen area (for example, *User 4* spends more than 90% of their time in windows that are typically less than 25% of the total screen area).

Screen usage corresponding to application behavior. In order to understand the relationship between the screen usage and the application behavior, we took the samples from each of our user logs and categorized them into four bins – (i) samples where the window of focus usage was between 0 and 25% of the total screen area, (ii) samples where the window of focus usage was between 25% and 50%, (iii) samples where the window of focus usage was between 50% and 75%, and (iv) samples where the window of focus usage was between 75% and 100%. For each bin, we then analyzed the key applications associated with the samples. Figure 4 summarizes our results. As before, we compute the arithmetic mean of the averages per individual users to avoid distortions due to trace lengths.

Overall, the workloads used by our user population span a range of applications representative of typical system usage. Broadly, they can be categorized into (i) access related - web browsing and e-mail (Internet Explorer, Netscape, Outlook, Pachyderm mail reader, Messenger), (ii) personal productivity and code development (Word, Emacs, Powerpoint, Excel, Visual studio, Dreamweaver, X-term, Realplayer, Image viewer, Acrobat reader, Ghostview), and (iii) system related and application control windows (File Explorer, navigation windows, taskbars, menus, status and properties messages, confirmation and password query windows).

Focusing on the windows associated with the various applications, we observe two interesting trends. First,

Active area is 0-25% *(23% of the time for typical user)*
Key applications: 20% Task bar, 15% Program Manager, 5% X-term, 60% miscellaneous windows (message composition, MSN messenger, real player, menu and message windows – properties, connection status, file downloads, alerts and reminders, volume control, printer status, find-and-replace, organizer preferences, file explorer, spell-check, wizards, status messages, file-find, password query windows, confirmation windows)
Active area is 25-50% *(22% of the time for typical user)*
Key applications: 19% X-term, 18% message composition, 6% internet explorer, 57% miscellaneous windows (mail-related windows, file explorer, emacs and notepad, MSN messenger chat windows, other status windows)
Active area is 50-75% *(28% of the time for typical user)*
Key applications: 33% Internet Explorer, 24% mail composition and reading, 43% miscellaneous windows (emacs and notepad, Image editor and photo viewer, messenger chat windows, Frontpage, Framemaker and ghostview, file explorer, Powerpoint, dreamweaver, winlogger)
Active area is 75-100% *(27% of the time for typical user)*
Key applications: 21% Outlook, 20% Internet Explorer, 7% Excel, 52% miscellaneous windows (Powerpoint, Framemaker, Acrobat reader, Word [various files], Visual C++ [various files]), Dreamweaver, Imageviewer)

Figure 4: *Understanding screen usage by application. Windows are classified based on their sizes into four bins, and for each of the four bins, the key applications dominating the samples in the bin are summarized.*

system-related status messages and query windows typically use small window sizes; in fact, these windows constitute a significant fraction of the samples associated with smaller size windows. Additionally, these windows usually display fairly low content that do not need the aggressive characteristics of the display – for example, a low-resolution display with support for a small number of colors would be adequate to obtain an equivalent user experience. Second, personal-productivity applications and development environments and web-browsing and e-mail applications typically use larger portions of the display area. The actual fraction of the screen area used appears to be highly dependent on individual user preferences for window size, fonts, etc. However, even with these large windows, characteristics of the displays such as resolution, brightness, and color are not used to their full capacity.

Screen usage corresponding to user behavior. Focusing on the individual user logs, we observe that individual user preferences and pre-set defaults tend to significantly influence the overall screen usage characteristics. For example, *User 1* who, on average uses 63% of the display area, has Internet Explorer set to use 96% of the screen area, while *User 5* who, on average uses 37% of the display area, has Internet Explorer set to use 67% of the display area. Similarly, *User 12*, who has the largest screen usage in our study, has a default mail composition window of 95% that dominates the traces. This user-specific sizing of windows appears to be particularly characteristic of web browsing, email, and editor applications. In contrast, for development applications (Visual Studio, and Dreamweaver, Powerpoint), most of our users prefer to have larger windows – possibly because of the multi-window content structure of these applications. Similarly, system-related and application-control messages typically use smaller windows irrespective of the user – mainly since the content in these windows is relatively low and in most cases the window sizes are pre-determined by the application. An illustrative examples is the case of *User 8* who maximized *all windows* as a matter of routine ("to be able to read better"). This user still consumes only 85% of the total screen area because of the smaller window sizes associated with system-related and application-control messages. Finally, while the laptop users have a slightly larger screen usage (62%) than the desktop users (56%), in general, the results are fairly similar over the laptop and desktop users.

2.3 Summary

Summarizing the results of our user study, overall, there is a significant mismatch between the properties supported in the display and the actual usage of these attributes by the users in our user study. The size of the display used exhibits the greatest mismatch – users use only about 60% of the screen area available. A large fraction of the smaller windows are typically associated with system-related and application-control windows that are independent of user preferences. User preference for smaller window sizes and font sizes can also translate into a greater use of smaller sized windows.

Similarly, there are significant mismatches between the actual screen usage and other attributes of the display such as resolution, brightness, color, refresh rate, etc. In particular, most of the smaller windows include content that could have been equivalently displayed, with no loss in visual quality, on much simpler lower-power displays (lower size, resolution, color, brightness, refresh rates, etc.) Many of the larger windows also do not use all the aggressive characteristics of the display.

Overall, these results indicate that energy-adaptive system designs that match display power consumption to the functionality required by the workload/user have significant potential to reduce the energy consumption of the display sub-system.

3 Energy-adaptive Display Sub-systems

This section studies some example energy-adaptive systems to evaluate how the potential benefits identified in the user study can be translated to energy reductions. Section 3.1 describes the hardware and software components of these designs. Section 3.2 discusses the experimental methodology used in prototyping and studying the user interfaces and the energy consumption for the different designs. Section 3.3 discusses our results and Section 3.4 summarizes.

3.1 System Design

Our designs use emerging display technologies and modified window system software to exploit the mismatches between workload/user requirements and display properties.

Hardware Support: OLED displays. To enable energy-adaptive designs, a key requirement is support at the display sub-system for variability in the display power based on the properties of the screen output. That is, it should be possible to change the energy consumption of regions of the display independent of each other. Several emerging display technologies support such variable power over different regions of the screen. This is preferable to existing technologies, where energy tradeoffs have to be made for the entire screen.

Organic Light Emitting Diode (OLED) displays [3] are a good example of this class. In OLEDs, the energy consumption of a pixel is related to its brightness and color. OLED displays are built from small organic molecules that efficiently emit light when stimulated by an electric field. More than 100 companies are developing aspects of OLED technology. Kodak, Sanyo, and Sony have shown prototypes from 5.5-inch displays to 13-inch displays at trade shows. In general, OLEDs have better image quality compared to conventional LCDs (better horizontal and vertical viewing angles, higher brightness, and faster response times) and do not need a separate backlight, resulting in lower power. As the technology matures, the biggest challenges are in overcoming yield problems and consequently reducing costs. Small OLED displays are currently available in devices like cell phones. Larger displays, for handhelds and laptops, are expected in 2004 [3].

For our prototype, we assumed a laptop system with a 15" AMOLED (Active Matrix Organic Light Emitting Diode) display. (Section 4 discusses other hardware approaches to implement the similar functionality.) The only hardware changes to a current commodity design will be to replace the conventional LCD panel, backlight, and controller with their OLED equivalents.

Software Support: *Dark Windows*. The software support for energy-adaptive displays can be implemented at one of several levels: the application level, the window system level, or the operating system level. In this paper, we focus on modifications to the windowing system. Specifically, the *"Dark-Windows"* modifications use the window of focus as an approximation of the active area of user interest to ensure that energy is spent mainly on portions of the screen that have the user's attention. For example, if a user is using an editor to take notes, her attention is typically on the screen area used by her editor, her current window of focus. Energy-aware changes to the brightness, color, or other properties of the remaining portions of the screen can result in energy benefits without impacting the user experience. Our discussions below focus on optimizations that automatically change the *luminescence* and *color* of non-active screen areas to reduce power. Section 4 discusses other properties of the background that can be changed and optimizations that focus on mismatches internal to the window of focus.

3.2 Prototyping and Evaluation Methodology

Our goals with the evaluation were two-fold. First, we wanted to understand the intrusiveness of the energy-adaptive user interfaces and the complexity and overhead associated with implementing the dark windows optimizations. Second, we also wanted to quantify the energy benefits in the context of one particular technology, namely OLED displays, and understand the design

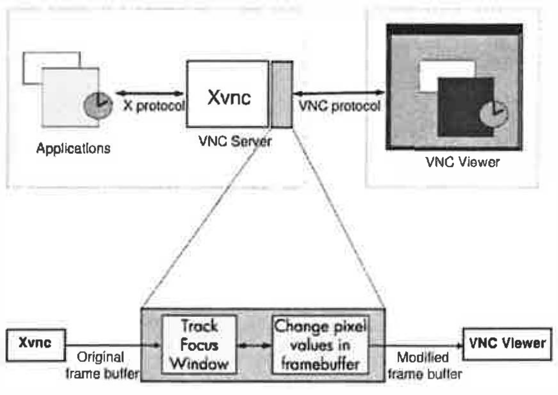


Figure 5: *Methodology used to prototype the user interface.*

Interface Name	Modification
HalfDimmed	Areas outside the window of focus are dimmed by 50%.
FullyDimmed	Areas outside the window of focus are fully dimmed (turned off).
GrayScale	Areas outside the window of focus are changed to gray, by setting red, green, and blue values to the average of the three.
GreenScale	Areas outside the window of focus are changed to green (lowest power color for OLEDs). The green value is set to the average of the three colors, and the red and blue values are zeroed. This also dims the screen by 67%.

Figure 6: *Summary of the* dark windows *optimizations considered. The window of focus is left untouched, while the background areas are modified in either brightness or color to save energy.*

tradeoffs between the various optimizations.

Prototyping the User Interface We implemented several prototypes in order to understand the impact of the dark-windows changes on the user interface. Given the difficulties with modifying the Microsoft windowing environments, we implemented our prototypes on the X Window System under Linux.

To enable a clean implementation of the changes, we used the open-source VNC (Virtual Network Computing) server [6]. VNC provides a virtual representation of the display hardware, i.e. a remote framebuffer, and allows one to run a server on a machine while viewing its display remotely from anywhere on the internet. While the remote viewing property was not of interest to us, by using VNC, we were able to gain two advantages. First, since it is a virtual frame buffer, we were able to easily manipulate the pixel values. Secondly, since VNC is linked with the X-Windows server, we had access to the window server data structures, such as the window of focus.

Figure 5 shows an overview of our changes. We modified the X-VNC server with two additional functions – to track the window of focus and other objects in the display area, and to change the values of the pixels in the frame buffer based on our dark-windows algorithms. To reduce the performance overhead from our software changes, our modifications closely track the incremental change update mechanism of the VNC protocol.[1] Each framebuffer update represents a region that has changed, and is sent to the viewer as required to keep the display up-to-date. Before each frame buffer update is sent to the viewer, we perform two functions. First, we query the X-server and record the window of focus – the window that keyboard events are directed to – including its origin, width, and height. We then divide the update region into two groups – those that are in the window of focus and those that are outside it. The pixels within the window of focus are sent unmodified, while the rest have the desired transformation applied to them. Overall, the performance overhead of our implementation was negligible with performance acceptable to users.

We experimented with several transformations, described in Figure 6. These included transformations that modified the brightness of the background (non-window-of-focus) regions (*HalfDimmed and FullyDimmed*[2]) as well as transformations that modified the color of the background regions (*GrayScale and GreenScale*). (Screenshots of these interfaces are shown in Figure 8 in the next section.) In all these cases, we also implemented a user-defined keyboard short-cut to be able to turn off the modifications if desired.

Quantifying energy benefits for typical users. Though we used VNC for implementation convenience, we still wanted to focus our benefits study on the commodity Microsoft Windows based systems used in our user study. We therefore constructed a synthetic trace that modeled the average behavior observed in our user study and re-

[1]We originally chose a simpler implementation approach where we simply traversed the finally-generated framebuffer to apply our modification heuristic on each pixel. However, this led to an inordinately large number of computations and caused a perceptible slowdown in the system. We chose to instead use the rectangular update mechanism to obviate this problem.

[2]Our optimization allows the brightness to be dimmed by any user-defined fraction; however, in this paper, for space reasons, we consider only two representative data points - half-dimmed and fully-dimmed.

Avg. window size	59%
Avg. background windows' size	17%
0-25% screen usage (taskbar, xterms, miscellaneous)	23%
25-50% screen usage (xterms, editors, mail readers, miscellaneous)	22%
50-75% screen usage (web browsers, mail readers, editors, miscellaneous)	28%
75-100% screen usage (web browsers, mail readers, miscellaneous)	27%
Default screen background	Teal
Default window color	White
Default foreground font	Black

Figure 7: *Summary of properties of synthetic trace. The trace models the average behavior exhibited in the user study and was created to match the data from Figure 4.*

played it on our prototype systems. The trace runs for about 1000 seconds and uses a set of applications similar in nature to those used in the user study. The script for the trace models the initiation and termination of applications and varies the window sizes and the duration of time windows were open to obtain statistics consistent with the user study. While the users in our test population used a variety of backgrounds and window colors, a majority of them used the default windows settings. Our synthetic trace therefore predominantly uses those colors. To bracket the impact of specific user choices in this respect, we also performed experiments where the screen background and window colors are varied. Figure 7 summarizes the window properties of the synthetic trace and a comparison of the data with that in Figure 4 shows the close correlation between the behavior of the trace and the user study.

Given the current unavailability of 15" OLED displays (aside from rare prototypes), we decided to use a software power model representative of future systems. The model computes the display sub-system power as the sum of the controller power, the driver power, and the display panel power. The controller and driver power are modeled as constant values irrespective of what is displayed on the screen. The panel power represents the pixel array power and is the total of the power consumed by each pixel in the panel array. In turn, the power consumed by each pixel is proportional to the brightness of the red, green, and blue components.

$$\begin{aligned}
&DisplayPower = P_{controller} + P_{driver} + PanelPower\\
&PanelPower = PixelArrayPower\\
&\quad = \Sigma P_{red} \times pixel_R + P_{green} \times pixel_G + P_{blue} \times pixel_B
\end{aligned}$$

The values of the parameters of the power model were chosen to represent likely future OLED displays, and were validated by discussions with researchers working in the area and by inspecting data sheets of current displays. For the 15" display we model, we set $P_{controller}$ to 0.5W and P_{driver} to 1W. The $PanelPower$ can be a maximum of 8.5W. Interestingly, with current OLED technologies, pixels consume different amounts of power based upon the particular color, with green typically being the lowest. For a 1024x768 resolution display with pixel values ranging from 0 (off) to 1 (fully on), the values for P_{red}, P_{green}, and P_{blue} are 4.3μW, 2.2μW, and 4.3μW.

3.3 User Experience and Energy Benefits

User Interface Intrusiveness. Figure 8 summarizes the various interfaces. (Color pictures are available in the electronic version of the paper.) The picture on the left represents the original configuration, while the two pictures on the top right represent dark-window configurations with modified background brightness, *HalfDimmed* and *FullyDimmed*. The two pictures on the bottom right represent the dark-window configurations with modified background colors, *GrayScale* and *GreenScale*. The performance overhead of the *dark windows* implementation was small and there was no significant difference in the response time of the user interfaces.

To determine the difference in the quality of the various interfaces, we informally surveyed 9 users. We asked each user two questions. First, *without describing any battery life issues*, we asked users to choose their best interface. Second, we showed the users the battery life advantages from the various dark-windows modifications and *then* asked users to choose their best interface again, with the assumption that they were in a situation that required long battery life, such as a meeting or an airplane trip. Most of the users indicated a willingness to use dark windows optimizations to tradeoff longer battery life for a different user interface. Of the 9 users we queried, 4 of them preferred *GreenScale*, 3 preferred *HalfDimmed*, and 2 preferred *FullyDimmed*. Most of the users expressed a desire to be able to see the contents of the background, even at the expense of some energy. Interestingly, even without an awareness of the energy benefits, four of the users chose some of the dark windows interfaces (*GrayScale* and *HalfDimmed*) as their preferred interface. Note that even without the software *dark windows'* optimizations, the base energy-adaptive display hardware still provides the ability to have lower power consumption based on what is displayed on the screen. Compared to LCDs, this can achieve some power

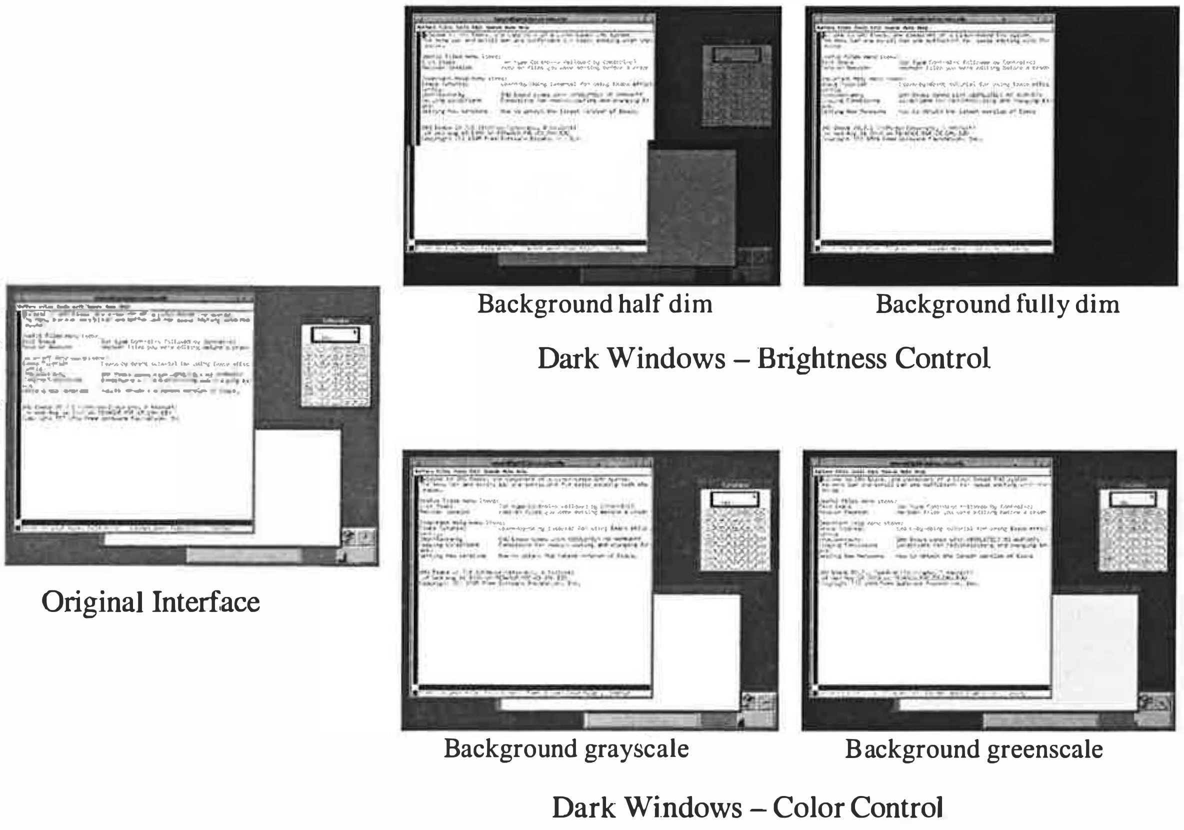


Figure 8: *Screen shots of the user interfaces with* dark windows *optimizations.*

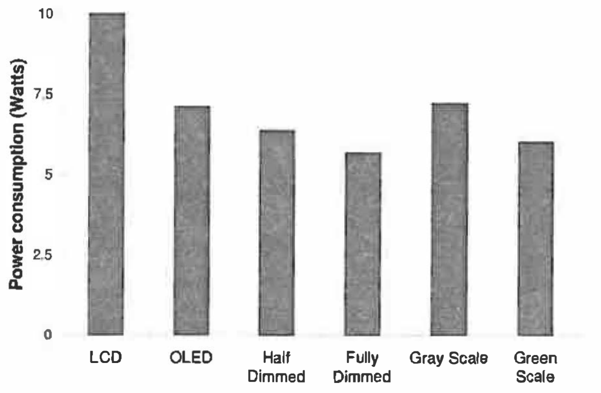


Figure 9: *Power benefits from energy-adaptive display designs.*

benefits with *no changes in the user interfaces.*

Our user interfaces could be improved in several ways (we discuss these in Section 4). However, our goal was not to perform an exhaustive exploration of the design space for user interfaces, but instead to establish that energy-adaptive display sub-systems can provide energy benefits with interfaces that users are likely to find acceptable, particularly in return for longer battery lifetimes.

Energy Measurements. Each of our interfaces modifies the pixels in a different manner, resulting in differing energy use. We measure the power consumed by our synthetic trace for each configuration on our modified VNC server. We study a default LCD configuration, a default OLED configuration, and four OLED configurations with the four dark windows optimizations discussed earlier. We use the OLED power model discussed earlier to compute the power for the five OLED configurations. The LCD power model is based on characteristics noted by Choi et al. [1]. For our synthetic trace, this leads to very little variance (less than 1%).

Figure 9 summarizes the results. Compared to the constant 10W power consumption of the LCD, the hardware support for energy-adaptive displays in the base *OLED* system achieves a 25% reduction in power. Most of these benefits are due to power reductions for the teal background color. (With our OLED display, a teal pixel - RGB: [0,131,131] consumes only 30% of the maximum power a pixel can consume - RGB: [255,255,255].) However, there are also some benefits from power reduction specific to the content of the windows (e.g., web browser). The software *dark windows* changes can provide additional power reductions. The *Fully-Dimmed* optimization provides an additional 20% over

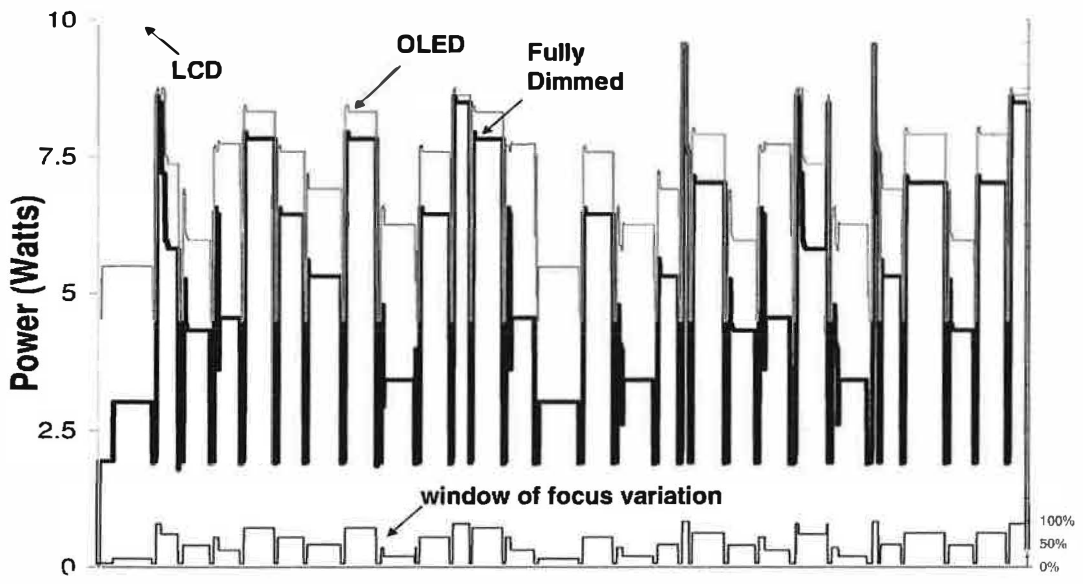


Figure 10: *Power variation over time for non-adaptive and energy-adaptive displays.*

the *OLED* case, and a total of 43% over the *LCD* case. These benefits are from dimming of the background windows (predominantly white) and the screen background (teal). *HalfDimmed* provides half these benefits while still allowing some of the information to be visible. The *GreenScale* optimization provides 40% energy benefits compared to *LCD* (and a 15% benefit compared to *OLED*). This optimization computes the average of the R, G, B values and includes it as the new value for the green pixel. (Recall that the green pixel takes the lowest energy compared to the other pixels.) The combination of using the most energy-efficient color and a reduction in brightness by 67% leads to the energy benefits. In contrast the *GrayScale* optimization averages the R, G, B values and uses this average as the new values for the R, G, and B pixels, turning them gray. However, our results indicate a 1% increase in energy with this configuration compared to *OLED* (though still 28% better than the non-adaptive *LCD* case). This is because converting our default background color (RGB: [0,131,131]) to gray scale ends up moving more bits to the higher-power R and B pixels (gray-RGB: [87,87,87]). An alternate background color such as pure blue or red may have benefited from having a grayscale background optimization.

Figure 10 shows the power variation over time for three cases - *LCD*, *OLED*, and *FullyDimmed*. The *LCD* case represents an non-energy-adaptive display and shows relatively constant power consumption, invariant to the size and content of what is displayed on the screen. The *OLED* case shows the benefits that can be obtained from using an energy-adaptive display technology. Since the benefits are mainly from the background screen color, the profile of the curve follows the profile of the percentage of the screen devoted to the background (after removing the window of focus and other background windows). The *FullyDimmed* shows the benefits from both hardware and software changes for energy adaptivity. The profile of this curve follows the profile of the window of focus curve.

As is evident from the above discussion, the benefits from energy-adaptive display designs are highly dependent on the choices of the background color and the window color. Our default trace modeled the windows default since that most closely represented a majority of the users in our test population. To better bracket the impact of other choices of background and window colors, Figure 11 shows the energy consumption of the various configurations for the extreme cases of pure white and pure black backgrounds and window colors. With black backgrounds and black windows, the base *OLED* design achieves most of the benefits from energy adaptivity – close to 80% reductions compared to *LCD*. The power consumed is mainly due to the controller and driver and minor elements of windows such as title bars, etc. Additional software optimizations get minimal improvements on top of this. In contrast, with white backgrounds and white windows, the base *OLED* configuration gets close to no improvements. The *FullyDimmed* system gets the maximum benefits (35%) by reducing the power spent on the background. The other two cases show intermediate points where both the hardware and software optimizations obtain good benefits. These results indicate

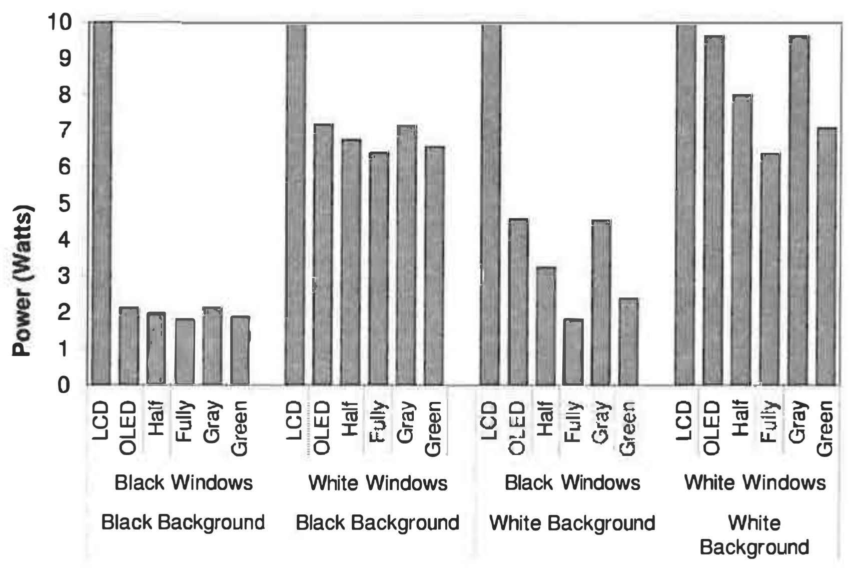


Figure 11: *Sensitivity of benefits from energy-adaptive designs to background and window colors.*

that different designs may be better for different usage scenarios. It is important for designers to understand typical user behavior before designing energy adaptive display sub systems. In some cases, it might be adequate to just include support for adaptivity at the display hardware level and choose color and size defaults with power in mind. In general, however, for systems that allow a great flexibility in user choices for background and window colors, it might be preferable to implement software optimizations in addition to the base hardware support for adaptivity.

3.4 Summary

Overall, our results indicate significant energy benefits from energy-adaptive display designs. The base OLED design achieves 30% reduction in energy compared to a base LCD non-energy-adaptive design – with no change in the user interface. The other dark windows optimizations change the user interface in different ways by dimming or changing the color of the background screen area and achieve significant, but user-specific, energy benefits. In particular, the choice of the background and window color can have a key impact on the power reductions. For the default windows background used by our users in the user study, the best optimizations, *Fully-Dimmed* and *GreenScale* achieve close to 40% energy benefits over the base non energy-adaptive design. An informal user study indicates reasonable acceptance of these user interfaces, particularly in the context of an awareness of the energy benefits from trading off the interface for longer battery lives.

4 Discussion

The configurations discussed in the previous section illustrate some example energy-adaptive designs. In this section, we discuss other possible energy-adaptive options for display design.

Display energy adaptivity in hardware. Rather than using OLEDs, other display technologies that enable energy-adaptivity can be used. These include other opto-electronic and emissive displays (Field-Emission Displays [FEDs] and even conventional Cathode-Ray-Tube [CRT] displays) as well as hybrid technologies like LCD displays with OLED backlights. With display technologies like LCDs that do not support energy variability, designs can still integrate a multi-modal "hierarchy of displays" configuration. For example, a mobile device could have two displays – one higher quality (high resolution, color, high refresh rate, larger size) and consequently higher energy use, and another lower quality and lower energy. While the adaptivity in this design is more coarse-grained than with the OLED systems considered in this paper, the insights from our study are still likely to be valid.

Display energy adaptivity in software. Rather than using the window of focus, other indications of user activity could be used. For example, the area around the

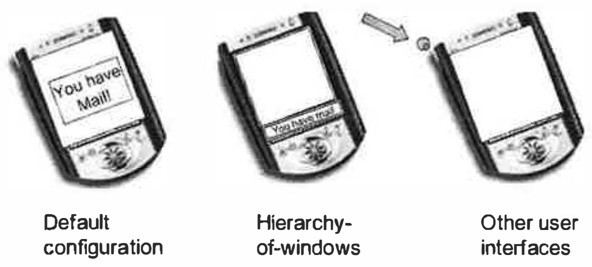


Figure 12: *Other energy-adaptive designs. The picture on the left illustrates a base system. The system in the middle uses two display panels to provide coarse-grain adaptivity for power. The right picture shows how display power for low-content messages can be reduced by using simpler visual or non-visual cues.*

cursor could be kept bright, while the rest of the screen is dimmed. Another design dimension is to include support for user-controlled dimming areas. For example, one possible user interface could include a *"sticky-lamp"* placed by the user to light up a specific portion of the screen. Much as we do in the physical world, the user could use multiple "sticky-lamps" to light up the work-area. An alternate implementation could include a "head-light" on the mouse pointer. The users could then point the light over regions that they are interested in and move down as they read along.

Still other dimming interfaces could be application specific. For instance, in a programming environment, there may be a concept of the current procedure and related variables. Portions of the screen related to these could be made bright – for example, all uses of the variables and all calls to the procedure. In an email application, perhaps only the current message needs to be bright. In a word processor, the line of text being edited could be bright, the surrounding couple of lines lightly dimmed, and the rest of the document greatly dimmed. Similarly, in the case of applications like Microsoft Powerpoint which use frames within an application, the notion of a frame-of-focus can be defined, similar to the window of focus. As another dimension to these user interfaces, these can all be made time-based. For example, areas of the screen that have recently changed could be bright, fading to a dim value as time progresses. When inactivity is detected, an email application could dim its screen area until new mail arrives.

Other user interfaces can be developed by combining the interfaces above. Additionally, other sorts of display mismatches could be exploited. In this paper, we have focused on identifying the mismatch between the total area of the display and the area of interest to the user. Other properties of the display, such as resolution and refresh rate could be exploited as well.

Support for output modes beyond displays. The notion of having multiple displays can be taken one step further to match output content to notification mechanisms beyond displays. For example, an email notification that says "You have mail" on the display could be replaced by an LED that blinks on the arrival of email or other similar notification mechanism such as speech output, vibrations, etc.(Figure 12). As discussed in Section 2, a lot of the smaller windows are typically low-content windows which may be amenable to other forms of non-visual communication. This combined with the large design space for alternatives for energy adaptiveness indicate the potential for an interesting future area of research – *energy-aware user interfaces.*

5 Related Work

Concurrent with our work, there are two other studies that have looked at adapting the output of the displays from an energy perspective. Choi et al. [1] perform a detailed characterization of the power consumption of the display sub-system of a handheld device (including the power of the panel, the panel bus, the backlight, the frame buffer and the data and address bus driving the frame buffer). They propose three optimizations that (1) vary the refresh rate to exploit the after-image caused by the time constant of the storage capacitors, (2) vary the color depth to be able to reduce the memory requirements and hence the memory power, and (3) vary the backlight luminance with a corresponding compensation of the brightness and contrast. Kamijoh et al [4] discuss the energy trade-offs in the IBM wristwatch computer. While they focus mainly on the hardware-level tradeoffs and kernel optimizations to use the various standby and idle configurations of hardware, they also discuss the implication of controlling the number of pixels turned on or off on the energy as well as reducing the duty factor of the display to control the brightness of the entire screen (e.g., at night). Additionally, while not focusing mainly on the display component of the power, Flinn and Satyanarayan [2] also evaluated in detail the energy benefit of reduced computation with lowered fidelity of images for web browsing and video playing. Their study also proposed a method called "zoned backlighting," to enable energy benefits in the display subsystem. While zoned backlighting could allow independent control of illumination level for different regions of the screen, no existing display supports such zoned backlighting yet.

Similar to these studies, our work also explores the possibility of adapting the output of the displays from an energy perspective. However, in contrast to the studies, we primarily focus on the *content* and *intent* of the output screen display. In particular, our work is the first to perform a detailed characterization of display usage patterns to identify and understand the common mismatches between workload/user needs and current display properties. Based on these insights, our work also explores several new user interfaces and hardware designs that allow energy-adaptive control on the portions of the screen that are not of immediate relevance to the user, while continuing to provide similar functionality on portions of the screen of relevance to the users.

Finally, this work focuses mainly on the display panel power. Complementary to our work, other studies have also focused on the power consumed by the display controller and driver (e.g., [8]).

6 Conclusions

As mobile systems, applications, and services become more pervasive, it becomes ever more important to identify design strategies to lower energy consumption and increase battery lifetimes. This paper focuses on the display sub-system and motivates and evaluates energy-adaptive system designs for future mobile environments.

The first part of our study performs a detailed analysis of display usage traces from 17 users, representing a few hundreds of hours of active usage. To the best of our knowledge, this study is the first to identify, quantify, and analyze potential mismatch opportunities in workload/user needs and current display properties. We find that on average, the window of focus – a good indication of the area of interest to the user – uses only about 60% of the total screen area. Additionally, in many cases, the screen usage is associated with content that could have been equivalently displayed, with no loss in visual quality, on much simpler lower power displays. Our analysis of the user traces indicates that many of these mismatches could be traced back to the typical content of the windows as opposed to specific user preferences.

Based on these insights, the second part of the study proposes energy-adaptive display systems that match energy use to the functionality required by the workload/user to obtain significant energy savings. To support the energy adaptivity in hardware, our designs leverage emerging display technologies like OLEDs that provide variability in the energy consumed based on the properties of the pixels. For energy adaptivity at the software level, we propose several *dark windows* optimizations that allow the windowing environment to change the brightness and color of portions of the screen that are not of interest to the user. We develop prototypes of the user interfaces and model the power benefits of such energy adaptive designs. Our results indicate significant energy reductions with acceptable tradeoffs in the user interface.

In addition to the designs that we evaluate, we also discuss other points in the design space including several other alternative hardware and software interfaces for energy-adaptivity. These designs are likely to achieve even further energy benefits. Our work leads to several interesting challenges including the design of energy-aware user interfaces as well as more intelligent heuristics to automatically identify mismatches between the workload/user intent and the display sub-system functionality.

As mobile systems continue to develop, the contribution of the display power to the total mobile system power is only likely to increase, particularly in larger mobile devices. Similarly, as mobile workloads continue to develop, mismatches between the display system required for the most aggressive application and the needs of the common-case workloads are only likely to get exacerbated. The combination of these trends indicate a huge potential for system designs that flexibly adapt their energy consumption based on workloads needs. We believe that energy-adaptive display designs like the ones that we discuss in this paper are an extremely promising approach to exploit this potential and that they will become an important part of future mobile system designs.

7 Acknowledgements

This work greatly benefited from discussions with Ramakrishna Anne, Bill Hamburgen, John Rudin, and Ken Nicholas. We would also like to thank Norm Jouppi, Dan Sieworek, Amin Vahdat, and the anonymous reviewers for their comments on the paper. We would like to thank John Sontag and Jim Rowson for their support of this work. Finally, we would like to thank the users who participated in our user study. Lu (Annie) Luo's work in this paper was primarily supported by an HP internship. Her research is also funded in part by NSF under award number CCR-0205266 and by an IBM PhD Fellowship.

References

[1] Inseok Choi, Hojun Shim, and Naehyuck Chang. Low-Power Color TFT LCD Display for Hand-Held Embedded Systems. In *Proceedings of the International Symposium on Low Power Electronics and Devices*, pages 112–117, August 2002.

[2] Jason Flinn and M. Satyanarayanan. Energy-Aware Adaptation for Mobile Applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, December 1999.

[3] Stanford Resources Inc., editor. *Organic Light-Emitting Diode Displays: Annual Display Industry Report*. Second edition, 2001.

[4] N. Kamijoh, T. Inoue, C. M. Olsen, M. T. Raghunath, and C. Narayanaswami. Energy Trade-offs in the IBM Wristwatch Computer. In *Proceedings of the Fifth International Symposium on Wearable Computers (ISWC-01)*, pages 133–140, October 2001.

[5] Trevor Pering, Tom Burd, and Robert Broderson. The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms. In *Proceedings of the International Symposium on Low Power Electronics and Design*, August 1998.

[6] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual Network Computing. In *IEEE Internet Computing, Vol.2 No.1*, pages 33–38, Jan/Feb 1998.

[7] Sanjay Udani and Jonathan Smith. The Power Broker: Intelligent Power Management for Mobile Computers. Technical Report MS-CIS-96-12, Distributed Systems Laboratory, Department of Computer Information Science, University of Pennslyvania, 1996.

[8] S. Xiong, W. Xie, Y. Zhao, J. Wang, E. Liu, and C. Wu. A Simple and Flexible Driver for OLED. In *Proceedings of the Asian Symposium on Information Display (ASID99)*, pages 147–150, 1999.

iMASH: Interactive Mobile Application Session Handoff

R. Bagrodia[*], S. Bhattacharyya, F. Cheng, S. Gerding, G. Glazer,
R. Guy, Z. Ji, J. Lin, T. Phan, E. Skow, M. Varshney, G. Zorpas
UCLA Computer Science, Los Angeles, CA 90095-1596

Abstract

Mobile computing research has often focused on untethering an in-use computing device, rather than enabling the mc bility of the computation task itself. This paper presents an architecture, implementation, and experimental evidenc that together validate a new continuous computing concept, application session handoff. The iMASH architectu leverages previous work on proxies, content adaptation, and client awareness to provide a unique, middleware-enable capability for continuous computing. Implementation in both socket- and RPC-based environments shows that ver fast, secure session handoff of non-trivial client/server applications across heterogeneous client devices and network is feasible: experiments on a number of applications yielded handoff latencies ranging from 0.5s to 2s.

1 Introduction

July, 2005

Maria Salas, eight months pregnant, is injured in a car accident. Jim Brown stops to help and seeing Maria, he uses his wristwatch to call 911 while sprinting back to his car to get his broadband wireless PDA. Jim transfers his conversation from watch to PDA as he returns to Maria, since the PDA has bi-directional slow-scan video support.

The PDA also has a personal area network transceiver, which is used to discover and communicate with Maria's medical alert chip. This data is forwarded to the 911 operator, who uses it to identify and notify Maria's physician.

Dr. Hughes is out walking in the nearby mountains when her watch chimes urgently. Glancing at the display, she sees "patient injured," pulls out her PDA and instantaneously transfers the messaging session to the PDA. A PDA-based application retrieves medical history data about Maria as Dr. Hughes jogs back to her car.

As she begins driving, her PDA session is transferred to her car computer, which has a powerful processor, display, and voice-recognition. This system retrieves from Maria's medical records a range of image and video data that exceeded the PDA's capabilities. As she enters the city limits, network support is transparently switched from the rural wide-area network provider to a metropolitan-area network provider with higher bandwidth. The Medical Center computing infrastructure senses the switchover, and upgrades the quality of the image data it is sending.

Dr. Hughes joins a video-conferencing session with the 911 operator, emergency room, Jim, and en route rescue personnel. Guided by the emergency room and Dr. Hughes, Jim is able to apply limited, but situation-specific life support measures until the rescue team arrives.

As Dr. Hughes enters the Medical Center parking garage, her session transfers back to her PDA, which has a high bandwidth connection available on campus. She remains in contact and while waiting for the elevator, she consults further chart details and enters some additional notes. As she enters the emergency room, she transfers her PDA session to a system with wall-mounted displays so the trauma team can easily see it. The transferred session includes partially completed progress notes not yet formally saved in the patient records database.

Maria soon arrives, and the trauma team and Dr. Hughes are ready for her: they already know most of what they need to do to heal her and save her unborn child. In fact, the rescue team has already begun critical steps under the joint guidance of the trauma team and Dr. Hughes.

Although much of the technology implied in the scenario exists today, either as off-the-shelf products or research prototypes, a critical missing link is software support for *continuous computing*, which would enable a session to move seamlessly between heterogeneous platforms. A primary goal of the iMASH project is to develop the concepts, architecture, and prototype implementation to provide effective support for such a service. A key enabler is *application session handoff* which leverages existing work on proxies, content adaptation and client-awareness to provide a unique capability for continuous computing that satisfies the problem constraints outlined below.

The core concept, challenge and claim of application session handoff is that it is possible to easily identify a subset of application state that concisely captures the se-

[*]This research is supported by NSF Grant ANI-9986679. Contact authors at {rajive, sbhattac, fred, sbg, glenn, rguy, jizr, jinsong, phantom, eskow, maneesh, zorpasg}@cs.ucla.edu

mantics of a partially completed computation; efficiently transfer that state subset to a different client device; and effectively resume work on that target platform—all with very low latency while satisfying appropriate security, scalability and heterogeneity constraints.

Each of the above three steps is a challenge in its own right. Our main effort to date has been on the second challenge of efficiently transferring state to another, possibly heterogeneous client device, and that focus is reflected below. The key insight in our approach is that object type knowledge can be exploited to manage the amount of state that needs to be transferred between clients, in much the same manner that proxy transcoding manages the amount of state delivered from server to client.

In this paper we outline application session handoff, detail a supporting architecture and its implementation, and validate the concept with experimental evidence.

1.1 Problem Characteristics

The continuous computing problem space is quite large and diverse. Our particular problem domain, medical informatics, displays a multidimensional character that constrains the solution space in a number of ways:

client/server computing: Much of medical informatics today uses this model; other models (e.g., peer-to-peer) clearly warrant future research.

heterogeneous client devices: Up to six orders of magnitude routinely distinguish client devices in dimensions such as CPU speed, memory, secondary storage, display area, input ports, and network bandwidth and latency. The permutations of dynamic conditions over these ranges effectively yields enormous heterogeneity.

heterogeneous network infrastructure: In a hospital, most intranet infrastructure can be provisioned with high-capacity, low-latency components, but the last hop characteristics will likely vary widely.

user mobility: Highly mobile users traverse a range of settings in which physical heterogeneity imposes diverse constraints on computation and/or communication capabilities, in turn motivating a user to dynamically (and unpredictably) choose from a spectrum of client platforms. This implies that content adaptation is essential: data delivered from server to client should be tailored to satisfy constraints imposed by current network conditions and device characteristics.[FGBA96]

Mobility further introduces a new problem: the need for in-progress computation and communications to move across heterogeneous client platforms with low latency. The combination of movement, diversity, and timeliness has been an unsolved problem.

legacy servers: Medical informatics administrators are conservative about "enhancements" to their server environments (cf. [vMCT95]). Any novelties we propose need to leave a legacy server largely untouched: no additional code may be added to the server software or even share the processor, and no changes in the semantics of client/server interaction are allowed.

client application-aware: It is acceptable for client applications to be aware of changes to the legacy environment. In the spirit of Odyssey[Nob97], we believe that application awareness can be beneficial to both the application and the enhanced system environment. Client awareness implies that the application be amenable to augmentation: source code is available, and semantic knowledge is accessible.

diverse data types: Medical databases contain high-resolution still image series (gray-scale, false-color, full-color), gray-scale medium-resolution video, dictation audio, and traditional text-based records.

These domain-driven characteristics have shaped the development of application session handoff and the resulting iMASH architecture. Our work is applicable to other domains to the extent that they generally share the same criteria; a change of constraints might enable (or preclude) different architectural choices.

1.2 Related work

Our work builds on several well-known areas (proxies, content adaptation/transcoding, client-awareness) and is complementary to recent work on continuous computing. The seminal work on proxy-based content adaptation is the BARWAN project[FGBA96], in which the argument for adapting data on-the-fly is made. Recent work by Lum and Lau [LL02a, LL02b] focuses on the decision-making aspects of content adaptation, including both the complexity of making such decisions, and the trade-offs between on-the-fly approaches and storing pre-adapted version of data. The benefits of client-awareness of environmental changes in Odyssey are presented in [Nob97].

Relevant recent work in continuous computing includes the One.World project at the University of Washington [GDLB02], which looks at the pervasive application migration problem from the program development environment, and proposes a structured view of communication channels that requires the explicit disconnection and reconnection of servers. (iMASH very deliberately preserves open channels from a server's perspective; our work at the application level contrasts with the approaches of Snoeren [SB00] and Zandy [ZM02] at the network level.)

One.World also argues for a homogeneous execution platform, to support the movement of code, where we presume code is already resident on a device. An exhaustive treatment of conventional process migration is found in [MD00]; our focus on heterogeneity precludes the direct application of these techniques. Roman et al. [RKC01] propose the use of reflective middleware to support continuous computing across heterogeneous platforms.

Early portions of iMASH research have been presented at a handful of workshops [PGB01] [LGGB02] [PZB02] [SKP02] ; this paper provides an overall integrating context for that work, and presents new experimental data based on an implementation of the entire architecture.

1.3 Road map

The next section provides an overview of application session handoff (ASH). Section 3 describes an architectural design that implements iMASH handoff. The architecture is followed by a report in Section 4 on our experience with iMASH, in the form of a series of experiments on various iMASH-enabled applications. Conclusions and future work are presented in Section 5.

2 Application Session Handoff

Application session handoff addresses all of the characteristics delineated in Section 1.1. The most challenging of these are heterogeneous client devices and networks, especially when poorly provisioned in these regards. Impoverished clients are unable to use emulation to execute foreign tasks and low-bandwidth network links can not deliver large data objects or sizable binary process images in a timely fashion. The ASH approach looks through the opaque shroud that traditionally envelops an executing process, so that it can apply needed conversions to essential state that allow for the computation to move to and continue on the target device. iMASH enlists the assistance of the application itself to identify essential application state, a step we call *semantic savepointing.* The state is transferred to an intermediate host, where conversions are applied to hide the heterogeneity between source and target clients. The converted state is then delivered to the target client, where it is used by a target-native version of the application to "restart" where computation paused on the source client.

This client-aware approach is surprisingly well supported by existing applications—in particular, those that savepoint state for recovery (e.g., Microsoft Word 2001). Such an application is already structured with preservation of essential state as a first order concern, and also is prepared to resume execution based on well-formed state. This issue is explored further below. Modern applications are also commonly re-targeted to heterogeneous platforms; versions of the Microsoft Office suite, for example, have been created for Windows NT workstations and PocketPC PDAs. Note further that content adaptation is increasingly common, though typically on data delivered from server to client; content adaptation as an integral feature of handoff is novel.

In client/server environments, the bulk of data used by a client application is provided by server(s). ASH provides proxy-based content adaptation to ensure that data delivered to a client is compatible with client constraints, and arrives in a timely fashion. Content adaptation is driven by both relatively static user, device, and application profiles, and typically dynamic network profiles. A proxy is employed to shield legacy servers from change, and to relieve potentially weak or overburdened clients from the complexities and costs of content adaptation.

ASH anticipates narrow last-hop network links in the upstream direction as well, and so it exploits caching at the proxy to eliminate the need to move server-supplied data *from* the client. Newly created data, of course, must be transferred explicitly. ASH also incorporates, with the assistance of the application, a facility that enables client operations on data to be concisely shipped and quickly re-applied to a cached data copy. Together, these two features enable the first half of ASH ("session suspend") to be accomplished with low latency.

The second half of ASH, session resume, involves the delivery of session state to the target client and initializing the target application. ASH satisfies the constraints of the target client and its network environment in part by content adapting the session state before delivery. ASH also supports the possibility that the target client application is sufficiently different from the source client version that it may only desire a subset of session state elements. Prior to state delivery to the target, an XML description of the state elements is given to the client, which in turn requests the session (sub)set of interest. It is this subset that is adapted and delivered.

Session resume, like session suspend, is designed to operate with low latency. Our experimental evidence (see Section 4) shows this to be the case, except when a poorly ported application requires excessive resources on a weak client—such as a large Java application on a slow PDA. In such situations, modest latency (<10s) handoff results.

A client application under consideration to be iMASH-enabled is required to be a suitable candidate for *semantic session savepointing*: it must be feasible for an application programmer to identify points in the application's execution at which the essential semantic state can be frozen. Interactive applications tend to naturally have such points, even when multi-threaded, as a side-effect of mechanisms that support user interaction. In many cases, application session handoff will be formally instigated by the user via the legacy interaction mechanism.

Note that in general, saving client state back to the server, terminating the local execution and starting an application on the target device (conventional "checkpoint/restart") is not acceptable: the client state may be incomplete, and thus either is not acceptable to the server, or is not appropriate to expose to other clients.

Freezing client/server interprocess communication (IPC) requires a bit of care. Discrete object transfers are straightforward (simply wait for the entire object to be delivered), but streaming data is more difficult: in general, handoff should occur at a "good" point in the stream, which is at the least a protocol-specific issue. For example, a reasonable point to freeze processing of a motion JPEG stream is at a frame boundary; but for an MPEG stream, a full "group of pictures" is the proper semantic unit boundary. A correct freeze may require cooperation between the client application and the session handoff service [LGGB02]. (See also Section 3.3.1.)

We have iMASH-enabled a variety of applications, including several web browsers, a paint program, a remote shell utility, a radiology teaching tool, and various video players. In some of these cases, source code was available but adequate documentation was not, which made the tasks of implementing semantic savepointing and session restart non-trivial. (For example, the remote shell had about 100 separate elements of session state.) In others, source code was *not* available, and yet we were able to substantially support application session handoff for these applications by creating a simple "wrapper" application for each that is itself iMASH-enabled and in turn executes the black-box application as a child process.

The Galeon browser (http://galeon.sourceforge.net), was iMASH-enabled in *under one hour* by using the wrapper technique. In this case, we leveraged Galeon's existing recovery mechanisms that aggressively maintain current application state (e.g., per-window history and onscreen window placement) in a set of XML-based files. The wrapper simply re-packages the files as session state during handoff, and the target side wrapper creates files expected by Galeon on startup. Additional time (about a day) was required to enhance the wrapper with an HTTP proxy capability so that Galeon's HTTP requests would route through iMASH, thus enabling iMASH content adaptation benefits. By leveraging iMASH's extensible content adaptation architecture (see Section 3.3.2), just one more day was invested in designing and building a content adapter for the XML-based state, so that automatic device-specific window resizing is performed during handoff, as shown in Figure 1.

iMASH-enabling an application could be greatly eased and possibly even fully automated with appropriate application programming language constructs and compiler hints/directives available to the programmer. This is a topic of ongoing research (cf. One.World [GDLB02]).

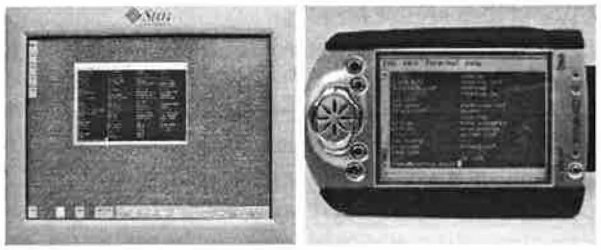


Figure 1: Pre- and post-handoff views of an application session handoff from a 19" workstation to a 4" PDA screen. Note the large application window has been resized to fit the smaller display.

3 iMASH Architecture

In our problem domain, the two most significant factors affecting architectural design decisions are client device heterogeneity and unaware (immutable) legacy servers. The former argues against an operating system-level architecture, to avoid repeated re-implementation for each operating system; the latter implies that a conventional middleware-level client/server solution is inadequate, because no place exists to host "server side" middleware. Further, clients are often ill-suited to hosting additional computation burdens.

The iMASH solution is middleware-oriented, but we place minimal middleware burden on clients—and none on legacy servers—by introducing a *middleware service* hosted by (new) additional platforms placed between clients and legacy servers. This service acts as a client proxy to legacy servers, and is in the critical path between clients—effectively a proxy between them.

The architecture material is organized in three parts: Section 3.1 motivates the need for multiple middleware servers to instantiate the overall middleware service; the resulting impact on application session handoff is then considered in Section 3.2; and in Section 3.3 we delve into the details of middleware server structure and operation.

3.1 Middleware service

In general, the middleware service must be reliable, secure, scalable, transparent to legacy servers, low latency, and impose low overhead on clients—all while providing its basic service, application session handoff. To achieve this, we believe that the middleware service must itself be a distributed system. The iMASH middleware service is provided by a collection of *middleware servers* that work cooperatively to support application session handoff. Figure 2 shows this structure.

A middleware server (MWS) mediates *all* communication between client and server, and it additionally maintains application session state on behalf of clients so as to assist with low-latency application session handoff between client devices. It also provides the bulk of the se-

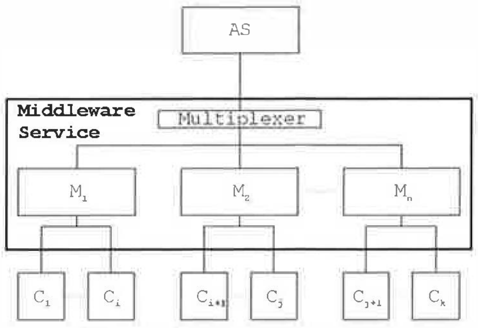


Figure 2: Middleware server architecture, with a number of middleware servers (M_n) on the middle tier and a multiplexer just above, and their current clients (C_k) on the lower tier.

curity services on which the above two functions rest.

3.1.1 Multiple middleware servers

The iMASH middleware service is routinely expected to perform potentially time- and space-expensive computation (e.g., adaptation, encryption, session state storage). With thousands of client devices and application sessions anticipated in a large hospital, it is reasonable to project the inadequacy of a single middleware server, and so a scalable architecture that allows a number of middleware servers is essential.

There are additional benefits to multiple middleware servers, if the system is properly designed. For example, multiple middleware servers also enable the deliberate placement of middleware servers at distinct points in a network topology, such as close to a wireless base station. One could also place a MWS at a remote location, so as to position adaptation closer to the client device and user. As a client device moves geographically, the "best" path between an application server and client may change and thus motivate a change of middleware server. Even when a client is stationary, the presence of network congestion or other conditions might spur a change of middleware: a sufficiently better path to the client device might exist from a different MWS, which could justify the cost of a change. The ability to change middleware servers on the fly further allows for dynamic load balancing (see [PGB01]) at both session admission and intra-session.

The need to maintain legacy application server unawareness of iMASH conflicts with the need to change middleware servers: the middleware proxy role hides the client application from the legacy server, but naturally substitutes its own visibility. iMASH deals with this problem by introducing a very thin service layer ("mux") which multiplexes communications between a legacy application server and middleware servers.

The mux has no role beyond hiding a change of middleware server from the unaware legacy server. From the application server's perspective, the mux *is* the client. Following instructions from the middleware servers, the mux merely directs (or redirects) a stream to the appropriate MWS. In our testbeds, we typically implement the mux as an application-level router on a conventional PC; there are obviously programmable router solutions that would be expected to incur much lower latency. If the legacy server operating system supports mobile sockets [ZM02], the mux could be eliminated entirely.

3.2 ASH types

To jointly support both multiple client devices and multiple middleware servers, iMASH provides application session handoff in two directions: client handoff, in which the device executing an application changes; and, middleware server handoff, in which the middleware server supporting the session changes. The resulting three handoff types are depicted in Figure 3.

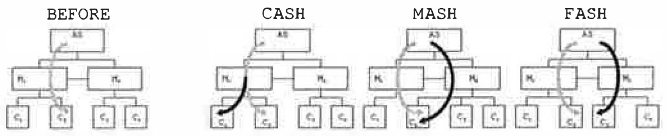


Figure 3: The three types of application session handoff. The gray arrow indicates a pre-handoff data flow from an Application Server (AS) through the middleware service to a Client (C). The black arrow shows the corresponding post-handoff relationship for the respective types of handoff.

We use CASH to denote a client-only application session handoff in which both source and target clients are supported by the same MWS. We use MASH to denote a middleware-only application session handoff, in which the client application continues execution on the same device while the session support switches to a different MWS.[1] We use FASH to denote a third type of handoff, full application session handoff, in which a new client and a new MWS support the session after handoff.

The motivation for CASH is driven by the need for a mobile user to change devices when moving among heterogeneous environments. CASH is almost always user-initiated since the user is switching physical devices. This implies that CASH generally occurs on a coarse time scale, with a typical frequency of minutes to days. Users may also be willing to tolerate a few seconds of handoff delay. In the opening scenario, the session transfer from PDA to car computer is an example of CASH.

MASH is driven by infrastructure concerns: scalable performance and flexibility in network topological and geographical placement, and responsiveness to network performance issues. MASH is rarely user-initiated, and

[1] MASH is loosely analogous to cellular telephony handoff between cell sites, but occurs at an application level quite independent of the network level.

so must appear to be delay-free. In the scenario, the WAN-to-MAN city limits switchover is a MASH.

FASH is much closer to CASH than MASH in both spirit and design: like CASH, it is driven by the user, and the change of middleware servers is an internal side effect. An example of FASH in the scenario is the session transfer from PDA to emergency room computer.

3.3 Middleware server architecture and operation

The iMASH middleware server architecture supports application session handoff through a variety of mechanisms, as diagrammed in Figure 4. Here we describe several of the most important ones, including session handoff management, content adaptation, and security.

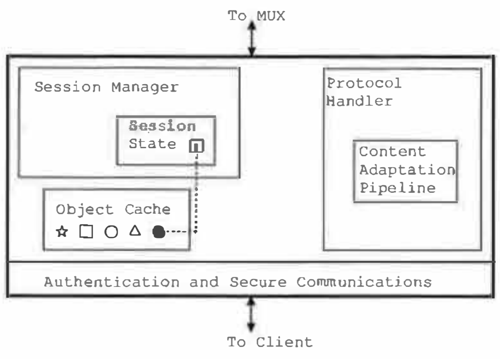


Figure 4: The middleware server architecture, showing the Session Manager, Object Cache, Protocol Handler, Content Adaptation Pipeline, and Security Layer.

3.3.1 Handoff Management

Session handoff management naturally divides between the types that require a semantic session savepoint (CASH and FASH), and the one that does not (MASH).

Savepoint-based Handoff Effective savepoint-based handoff relies heavily on the ability to exploit actions taken in the normal course of events that occur in client-to-MWS-to-legacy server interaction. In particular, iMASH caches at the MWS discrete (i.e., non-streaming) server-supplied objects for later use in session handoff.

Recall from the opening scenario Dr. Hughes' use of the PDA to retrieve Maria's medical records. An application on the PDA is executing and making queries of a legacy medical records server at the (remote) hospital. The middleware server mediating the connection recognizes that the requested medical record's complex data structure includes several data types for which the PDA is generally inadequate: high-resolution images whose size exceeds the PDA's memory, and video which exceeds the PDA's processor speed to display. None of the instances of these types are passed on to the PDA client in their original form; however, the images are cached at the MWS, and "thumbnails" (content-adapted versions) are delivered to the PDA, along with unmodified text-based data objects.

At this point, the PDA-based client application has received a mix of original and content-adapted data objects from the legacy server, via the MWS. Upon entering the car, Dr. Hughes initiated a CASH from the PDA to car computer—which happens to be different (in this case, more capable) in many key respects, especially CPU speed, memory capacity, and display size. Ideally, the transferred session will automatically incorporate data object versions appropriate to the target device, unbiased by the source device's limitations.[2]

To effect the handoff, the current essential application state from the PDA must be collected and shipped to the middleware server, after which the source client application is terminated. At the MWS, the session state is cached, content adapted as needed, and delivered to the target device for insertion in the target application.

In our architecture, the handoff request is forwarded by the iMASH-enabled application through the *iMASH client layer* support library (previously linked into the application) to the MWS currently serving the session. The MWS verifies that the handoff is authorized (see Section 3.3.3), and if so, arranges with the target client (and target MWS, if a FASH) to be prepared for session handoff. In particular, a session skeleton is prepared on the target client, and the target application begins execution and waits for the savepoint to arrive. The (source) MWS then invites the source client application to savepoint.

iMASH application state is divided into three types: *server objects*, i.e., those obtained from the legacy application server, such as an X-ray image; *application objects* created locally by the client application, such as text comments about the image; and *private objects* created by the application, such as a temporary buffer holding copies of the image and comments. An application savepoint only includes server objects and application objects: private objects are those not intended to be part of session state. Server objects are logically, not physically, part of the savepoint, as they are already stored at the MWS, having been cached there on the way from the application server to the client. Application progress is represented explicitly as an application object in the savepoint.

When the savepoint is complete—which might not be immediately, depending on application activity and semantics—it is transferred to the source client's MWS. At this point, the source client cleans up its local session state and exits, retaining no session knowledge at all.

[2]This *must be the case* to avoid a greatest common denominator effect, in which the least capable client constrains the session's future—and thus neuters the value of handoff.

The MWS merges the savepoint with any existing stored session state, and sends a summary list of session state elements to the target client. For FASH, the session state is first copied to the target MWS, which then finishes the handoff interaction with the target client.

The target client selects which elements of session state it wishes to receive, and makes an aggregate request to the MWS for them. The MWS sends copies of the selected session state elements to the target client after possible adaptation. When the target client has received all of the requested elements, it initializes its application-specific semantic state and "resumes" execution.

The extra hop through the middleware server is essential: clients are not presumed to have communications hardware and services in common; weak clients are not forced to deal with heterogeneity-induced data format conversions; and low-latency handoff requires an economy of upstream data movement which middleware servers can provide by caching objects and other application session state. The MWS maintains session savepoint data as a part of the session state—even after execution resumes at a target client device.

Storing a copy of server-supplied objects at the MWS is a key issue in providing very low latency application session handoff: while a number of large objects may have been received by the client application over a long period of time, handoff must be accomplished in a short interval. iMASH must avoid moving data from client to MWS wherever possible, since in many interesting scenarios the upstream channel is severely constrained.

Further, client heterogeneity often makes the version of an object at the source client uninteresting elsewhere: adaptation done to the object on its way from the server to the source client is client-specific. A very different adaptation probably should be performed for that object when destined (during ASH) to a heterogeneous client.

Middleware-only Handoff Middleware-only application session handoff is different than CASH and FASH because the client is typically not the initiator. Instead, the middleware server initiates the handoff, perhaps due to its awareness of a better path to the client through a different MWS, or perhaps because the current MWS is overloaded. This is a significant difference, because in CASH and FASH the client explicitly chooses when to do a handoff, and latency incurred at that point may be accepted by the client as the cost associated with the benefit. But in MASH, the user does not routinely participate in the middleware handoff decision, and so should not be noticeably penalized by handoff latency. We are exploring both passive (upcalls) and active (downcall, polling) approaches to communicate important changes in network conditions to iMASH. The middleware prototype responds to an external application-level network monitoring task that periodically assesses packet delay.

Once a MWS has decided that a MASH should occur, it contacts the target MWS and indicates its desire for a MASH. The target MWS has an opportunity to deny the handoff (perhaps it is overloaded), but in the successful case it prepares a skeleton session in anticipation of receiving session state from the source MWS. Upon receiving a positive acknowledgment from the target MWS, the source MWS notifies the mux that it should immediately redirect specific streams to the target MWS.

The mux knows nothing about the content of a stream flowing through it from the application server to a middleware server. Therefore, a stream is redirected at an arbitrary point. The target MWS must buffer the redirected stream, until it has received all session state from the source MWS. In addition to routine session state, the source MWS must also forward any buffered stream data that has not been processed, so that this data may be prepended at the target MWS to the front of its buffered stream data to preserve the correct stream data ordering.

3.3.2 Content Adaptation

Content adaptation has been well-trod ground for several years [FGBA96, Nob97, LL02a, LL02b]. iMASH exploits content adaptation to ensure that only client-device appropriate data is delivered to a client application, whether the data is coming directly from a legacy application server, or indirectly in response to application session handoff. The former case is analogous to standard proxy-based content adaptation [FGBA96]; the latter is an iMASH-unique variation which relies on a middleware server-cached copy of an original object as a basis for handoff-induced content adaptation.

The basic iMASH content adaptation architecture is fairly conventional: it includes tools that recognize ISO/OSI Layer 4 and above protocols, parse as needed to extract objects (possibly deconstructing complex aggregate objects), perform selected content adaptation, and reinsert the adapted object(s) into a stream of a type expected by the client application. The architecture is extensible, in that it anticipates new protocols and data types. It also allows for a series of adaptations on an object, much like a UNIX-style pipeline of filters.

Profile-driven Content Adaptation The general content adaptation problem is one of constraint satisfaction: given an object with certain characteristics, a set of constraints on those characteristics which must be met, and a set of adapters which can transform characteristics, which adaptation(s) should be applied in which order to minimally meet the constraints?

iMASH employs a three-step process—data characterization, command generation, and pipeline execution

[PZB02]—which today relies on simple heuristics to determine what adaptations to apply. Each of the three steps is designed for extensibility, so that additional data types and adaptation functions can be added readily.

Object characterization is performed by a type-specific function which provides an XML representation of an object's attributes. If an object has no matching characterization function, iMASH simply passes the opaque object on the client with no adaptation.[3]

Constraints are obtained from several sources such as client device, application, user, and network, and represented as XML profiles. Profiles are by nature extensible. A *client device profile* typically describes the processor speed, memory capacity, display dimensions, and nominal bandwidth of the currently-in-use network interface. A *user profile* often contains user preferences such as a "patience factor" to express the duration of a tolerable per-object latency. A *network profile* describes recent dynamic network conditions; ongoing work is examining a tight integration of network quality of service information with content adaptation, especially of streaming data. Profiles are merged within the pipeline, giving general precedence to the most-restrictive constraints, to produce the set of constraints to be satisfied. A profile can also explicitly indicate that *no* adaptation may be performed.

The object characteristics and constraints are then fed into the *command generator*, where they are compared in a pre-determined order, and adapters are chosen which can transform the object to satisfy the constraints. A sequence of adapters is then executed on the object, and the final resulting object is given to the protocol handler component for re-insertion into the protocol stream and ultimately, delivery to the client.[PZB02]

3.3.3 Security

iMASH enables and encourages a much larger number and variety of client devices than the legacy systems which it enhances, and many of these clients are expected to be mobile. Mutual authentication between middleware servers, and between clients and middleware servers, must be accomplished on a large scale and done with very low latency during handoff.

In the medical domain (and others), information privacy and integrity is essential. Recent U.S. legislation imposes significant potential liability on those who deal with medical data [Hel00]. Today's mobile computing environment relies heavily on wireless technology with well-known exploitable weaknesses (e.g., see [BGW01]), so it is prudent to ensure privacy and integrity above this level. We also want to limit exposure resulting from a stolen client device or compromised middleware server.

[3]This is a simple policy decision. Another alternative is to return a null object.

Further, the novel iMASH notion of application session handoff raises new security issues above the network layer: How will trust relationships be transfered and maintained during handoff? How to transfer/adapt session encryption keys to ensure privacy when transferring a session to a new device? Who is authorized to trigger a session handoff, and to (and from) what client devices? What policies should govern such authorization, and what components enforce them?

Basic Solutions iMASH uses a bi-level security framework, layering a user/session authentication and authorization over a device level authentication [SKP02]. As the first step in (re)joining an iMASH network, a client device performs a mutual authentication with a single middleware server.[4] The authentication protocol and all subsequent data communication use WTLS[WAP], which uses a certificate-based, public key authentication procedure followed by the use of secret key symmetric encryption to protect data flowing on possibly lossy transport services (e.g., UDP over wireless). Each device and MWS has a unique certificate issued from a trusted authority; the certificate is assumed to be stored in a tamper-proof container on the device. We assume our certificate authority is scalable, reliable, and accessible.

Because critical steps in the public key encryption methodology are very computationally expensive—up to ten seconds or more to encrypt a 1KB block on a modest PDA—iMASH is designed so that it only requires a client device layer authentication when the client first joins the network, which in a fully iMASH environment is at boot time.[5] When the authentication handshake is complete, a low-cost, secure, encrypted, channel exists between the parties that uses a symmetric encryption algorithm[6] key produced dynamically during the handshake.

This initial *device control channel* is used to enable extremely quick generation and exchange of additional keys which in turn are used to establish secure *session control channels* (one per session) and *session data channels* between a client application and a MWS used for mediating communication between client and legacy application server. Each session has a distinct session control channel based on a unique key, so that sessions are individually protected. iMASH creates new keys for each application-requested session data channel to minimize potential data exposure through the compromise of a single key.

Some clients may find that even relatively inexpensive symmetric key encryption is too costly to impose on all data transfers. iMASH allows a user to specify that null

[4]Initial MWS selection can be arbitrary; a "poor" choice can easily be rectified by a subsequent middleware handoff.

[5]Middleware servers are presumed to be powerful enough that occasional costly authentication in the critical path is tolerable.

[6]WTLS supports DES, 3DES, RC5, and IDEA.

encryption is acceptable for session data channel communication for non-sensitive data. All device control channel and session control channel communication is required to use strong encryption, however, because these channels are used to create keys for new session/data channels.

The user/session level authentication takes place during session creation: a password-based protocol is used to authenticate a user to the MWS, prior to the initialization and execution of a client application.

iMASH security policies require that authorization be given in several places. First, the MWSs have policy regarding which sessions may handoff to which devices (e.g., a MWS may disallow moving a session containing patient records to a public workstation). Also target devices reserve the right to refuse sessions, so that users may allow only their own sessions to be transfered to their own PDA.

Secure Handoff Session handoff poses interesting novel security challenges. First, at device boot time, a device is authenticated to a particular MWS. The handoff target device is authenticated to some MWS, which may be different than the MWS of the source client. In the case of middleware handoff, the client device is not in general authenticated to the target MWS. In both CASH and FASH, client devices do not trust each other. In all cases, there is no time to do a computationally expensive handshake to establish trust with the new MWS or client device. This handshake avoidance is especially critical if one of the clients is a low-end device.

iMASH exploits the relationships that a client trusts the MWS to which it is connected, and the MWSs trust each other. The fact that all clients have a secure device control channel (DCC) is also critical. The transitive trust relationship allows a client to trust a second MWS in MASH or transfer its session to a second device in CASH or FASH. The existing secure DCCs enable new encryption keys to be created for all control and data channels on any handoff. This ensures greater security in the event that a device (or middleware) is compromised: once a session moves from the affected host, that host has no knowledge of the encryption keys used post-handoff.

4 Experiments

We have developed a prototype implementation of our middleware-based application session handoff architecture, and have iMASH-enabled a number of applications. This section outlines our experimental testbed and then describes several experiments using a mix of iMASH-enabled applications and middleware service implementations. The first two experiments use a socket-based iMASH architecture implementation with browser and remote shell applications; the third experiment employs an RPC-based iMASH prototype and a video player application. The experimental results presented below demonstrate that application session handoff is a viable approach to very low latency continuous computing.

4.1 Experimental Testbed

The purpose of the experiments reported here is to further establish the validity of the iMASH approach to application session handoff with a full-featured implementation under stress. To do so, we created a testbed that enables us to do controlled assessment of various aspects of iMASH. The testbed is designed to be reasonably representative functionally of a small scale iMASH environment. It contains a single application server, a single multiplexer (mux), two middleware servers, and four clients, as shown in the architecture diagram in Figure 2.[7] All of the experimental results presented here were produced on this testbed.

The physical testbed is largely constructed from initially homogeneous components that are selectively reconfigured (hardware and/or software) to produce desired degrees of heterogeneity. While an "ideal" testbed would have a variety of heterogeneous hardware and software pieces, there is also value in underlying homogeneity: for example, we were able to quickly trace an unexpected transient latency variance to a particular machine simply by exchanging roles with an identical machine. (We replaced the unreliable box, and re-ran the experiments.)

Our base machines are Dell Inspiron 4000 and 8000 laptops running Pentium III processors at 800MHz with 256KB cache, 128MB RAM, and standard issue 20GB disks. Network support is provided by PCMCIA-based 3COM 10/100Mbps "575" ethernet cards nominally running at 100Mbps through a pair of stacked 3COM OfficeConnect Dual Speed 8 switching hubs.

Our operating system environment is a Redhat Linux 2.2.17-8 kernel with most daemons enabled with default parameters (sendmail, ftpd disabled). Our client applications, autotester and all iMASH components are currently written in Java; the base JVM is Blackdown 1.3.1.[8]

In addition, the testbed includes a single Compaq iPAQ 3670 PDA running a 200MHz ARM processor with 64MB internal memory (32MB RAM, 32MB flash); a 240x320 pixel, 12-bit color display; and, a PCMCIA-based 10Mbps wireless WaveLAN IEEE 802.11b card. The iPAQ runs a Familiar Linux 2.4.7 kernel, and Blackdown Java 1.3.1.

In all of the results presented here, the application server, mux, and middleware servers execute on indi-

[7]as shown in iMASH design and current implementation support an arbitrary number of all components. Ongoing work is using simulation to extrapolate predicted large scale behavior.

[8]We compile our Java code with the "green threads" option, in large part to avoid known scheduling interaction deficiencies between this JVM and Linux 2.2 kernels.

vidual machines as described above. This is consistent with the target domain scenario of well-provisioned back-end infrastructure. Note that the mux component in the testbed is relatively weak and thus imposes much higher latency than would a "real" mux implemented on a programmable router. We also expect a MWS to be a powerful machine, as adaptation is often CPU intensive.

The testbed is set up with four client machines: three Dell laptops and one iPAQ PDA. One of the laptops is configured as a "powerful workstation", with a 100Mbps network card and "large" full color screen.[9] A second laptop plays the role of mid-range desktop, with a 1,000x1,000 pixel full color display and 100MBps network. A third laptop represents a truly mobile laptop, with the same parameters as the mid-range desktop excepting a 56Kbps network connection. The iPAQ is the fourth client, and has the screen size, color depth, and bandwidth constraints listed above.

The first two of the three experiments additionally share a common high-level testing regimen: we designed an *autotester* to drive an iMASH-enabled application with a specified synthetic workload, networking environment, and handoff profile.[10] The autotester software is designed to exercise specific aspects of iMASH in a randomized, yet repeatable, manner. The autotester is script-driven, and causes iMASH sessions to be created on specific client devices with specific applications and synthetic workloads, and externally triggers session handoff to specific clients.

4.2 Minibrowser Experiment

Each iMASH testbed component—legacy application server, mux, middleware server, client, or autotester—is hosted on its own machine. Additionally, we wrote a specialized "minibrowser" application which at startup obtains a pre-determined workload script of objects to request along with inter-request delays.

The workload consists of randomized sequence of requests for each of 200 unique image files, ranging in size from 0.5KB to 1.25MB. The sizes are roughly evenly distributed over the entire range, with some tail heaviness toward the smaller sizes. The inter-request delay is Poisson distributed with a mean of 2 seconds. Once started, the minibrowser executes the workload without further direct interaction with the autotester. Upon completion, the client exits, and the iMASH session terminates.

Concurrent with the minibrowser execution of the object request workload, the autotester uses a randomized yet repeatable script to trigger application session handoff (either CASH, MASH, or FASH) of the minibrowser session. In the work presented here, handoffs are injected into the application session execution with an inter-handoff request delay of 10 seconds.

The client application is a simple Java-based object viewer which can request and then display JPEG images. The savepoint state provided by the source client during handoff is small: a reference to the current object being displayed, and a small amount (under 100 bytes) of additional state. The session state actually transferred to the target client is typically much larger: the currently displayed object will be retrieved from a MWS cache during handoff, content adapted as appropriate for the target client, and then delivered to that client.

We conducted two minibrowser experiments. The first compares the performance of an iMASH-enabled environment with the non-iMASH version of that system. The second studies the performance of handoff across a range of clients, handoff types, and object sizes.

4.2.1 non-iMASH vs iMASH performance

The utility of an iMASH-enabled client is determined in part by the impact of the iMASH infrastructure on "normal" activity: the user-visible performance of conventional client-server interaction must be comparable in both non-iMASH and iMASH-enabled environments.

To assess the impact of iMASH on normal activity, we conducted experiments to measure the latency experienced by a client when requesting objects of varying size from the server (using HTTP *get* operations). We also varied the client device, to understand the impact of heterogeneity. One experiment employed an iMASH environment with application server, multiplexer, single middleware server, and single client application and device. The second experiment (the non-iMASH case) used the same application server; the client application is an iMASH-free version of the iMASH-enabled application. In both experiments, identical sets of randomized object requests were used, and identical client devices were used.

Figure 5 shows the results of this experiment. The

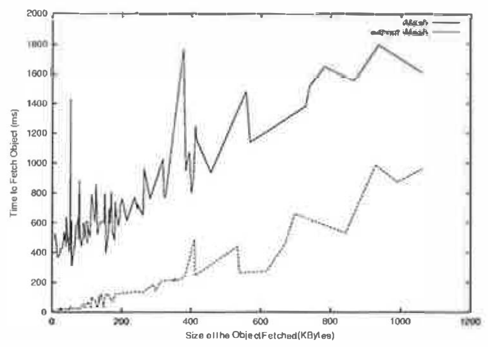


Figure 5: Average client latency experienced on object request (Y-axis), sorted by object size (X-axis), for iMASH and non-iMASH environments.

[9] "Large" means that no attempt will be made to content adapt an image to fit the screen

[10] This autotester is currently specific to the socket-based iMASH implementation, so it was not used for the video experiment.

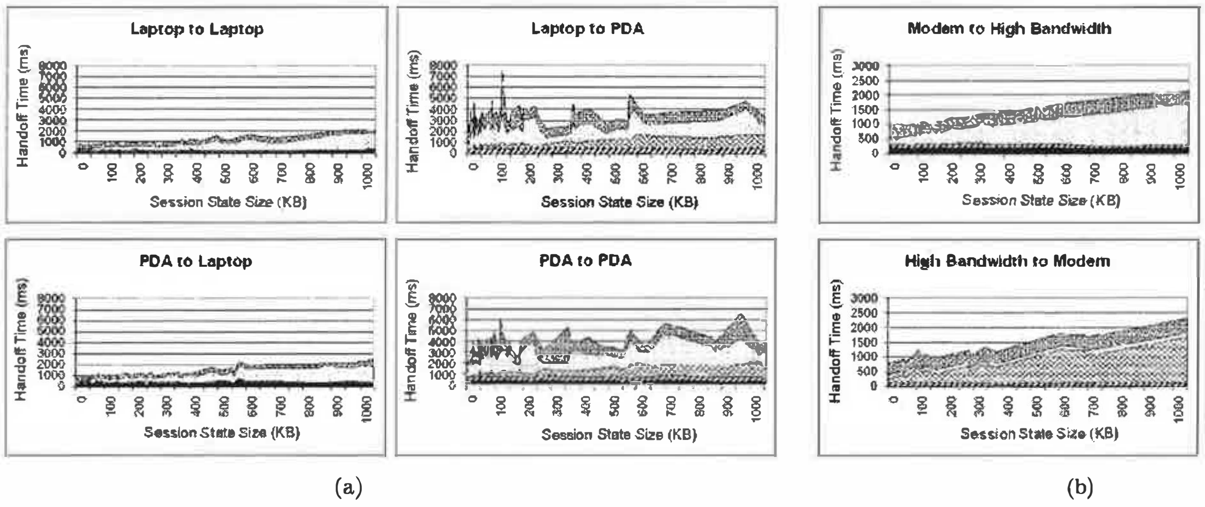


Figure 6: Client latency experienced on CASH (ms), as a function of session state size (KB). Note varying ranges on Y-axes. The upper curve represents the total handoff latency. Each band below the curve represents successive phases of handoff, from bottom to top.

■ state transfer from source client to middleware server
▨ initializing a skeleton session on the target client
▩ adapting the session state before delivery to the target
□ delivering the session state to the target client
▦ execution resumption on the target client

graph shows the average latency experienced by the application when requesting an object of a particular size. For brevity, we only show the "workstation" results. The graph shows that the latency increases linearly with the exception of a couple of outliers. The latency burden is about 0.5s for smaller objects, and approached 1s for larger object sizes. This baseline accounts for the additional costs of moving data up through a protocol stack on the middleware server, into a Java application, and back down through the stack to the client. The latency beyond the baseline 0.5s is largely due to the over-the-wire encryption incorporated by default by iMASH. We conclude that the additional latency of iMASH is tolerable even when no specific handoff benefits are exploited.

4.2.2 iMASH handoff performance

The purpose of this experiment is to determine the cost of application session handoff in terms of latency visible to the client application and therefore, the user. We also wish to understand the source(s) of such latency. In this experiment, a minibrowser application session was created and driven by an infinite loop over the 200 object randomized workload described above. Against this workload, a randomized series of handoffs was performed.

The graphs in Figure 6 summarize the results. They show the wall-clock latency (Y-axis) experienced by the application during handoff, with the session state primarily composed of the most-recently requested object, whose size is indicated. The data is separated by source and target client device type: Figure 6(a) shows data from handoffs involving a "full featured" laptop client and a wireless PDA client; Figure 6(b) shows data from handoffs between a wired 100Mbps laptop client and a 56K modem laptop client. Each data point is the alpha (= 0.10) mean of all occurrences of similar data.

The client device and network profiles for the PDA client and 56K modem client reflect the (greater) constraints faced in comparison with a typical wired workstation type of client. These constraints are sufficiently significant to cause the middleware server to invoke content adaptation in each case prior to delivering data to them; a small display size is the key profile constraint on the PDA, while slow network transmission is the prominent issue for the modem client.

Perhaps the most important observation is that handoff incurs a latency ranging from 0.5 to 7 seconds, depending on state size and target device. At a more detailed level, we see in these results that when the target requires content adaptation, a latency proportional to the source session state size is incurred. The time taken to deliver the state to the target is proportional to the source state size when no target constraints are in place, but when adaptation is invoked, this component is constant relative to the specific constraint. We also see that the time to resume execution is essentially a constant function of the CPU type—because the PDA is executing the same code as the laptops, it has a longer resumption delay.

From the same experiment that yielded the CASH results, we also extracted MASH results. They show that for session state sizes under 400KB, a nearly constant

210ms of latency is incurred to transfer a session from one MWS to another. For larger session sizes, a simple linear regression yields a latency increase of about 13ms per 100KB increase in state size—which closely matches the expected transmission delay over the 100Mbps network links employed in this experiment.

In Figure 7 we show corresponding FASH results. Because the effort involved in a FASH is somewhat similar to performing both a CASH and MASH, we expect to see slightly larger delays here—and, in fact, we see here the same basic trends as in the CASH results above, noting that in most cases a slight increase in the latency is present. This increase is largely attributable to the delay in copying session state from one MWS to another.

We conclude that the delays imposed by CASH, MASH, and FASH are indeed acceptable: system-initiated handoffs (MASH) are well under a second, even for large session states, and thus are unlikely to be noticed by users; user-initiated handoffs are generally under two seconds.

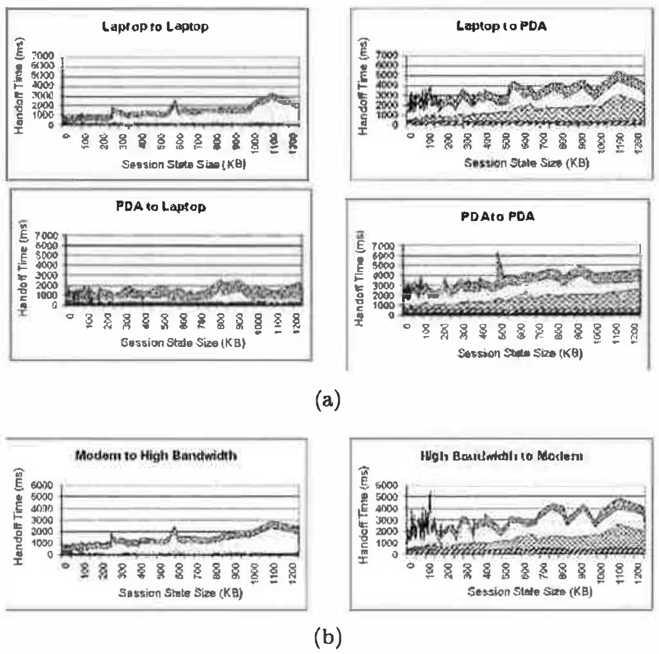


Figure 7: Client latency experienced on FASH, as a function of session state size. The X-axes units are bytes, ranging from ≈ 1KB to ≈ 1.3MB; the Y-axis units are milliseconds. The upper curve represents total handoff latency.

4.3 Remote shell experiment

The Java Telnet Application, or JTA [JM] is a Java based remote login client. The JTA emulates a VT320 terminal and can be used to connect to either a telnet or an SSH server. It was selected for integration within the iMASH infrastructure primarily due to the usefulness of having such an application with handoff capabilities, in addition to the desire to test the iMASH architecture with an application possessing substantial state. The JTA must save approximately 100 individual pieces of state during handoff, which together often total 67KB. The largest single piece of state is the window buffer; it contains the 100 most recent lines displayed on the terminal and can grow in size to as much as 32KB. In addition to its significant amount of state, the JTA itself is a relatively large application, and has a source code base which exceeds 16,000 lines of code. Although this code already resides on the target, it must be loaded by the JVM at invocation.

We employed the same autotester as used in the mini-browser experiment, but with a workload appropriate to a remote shell application: executing various commands that generated varying amounts of "stdout" output, and randomly performing handoffs. Figure 8 shows a histogram of the latency experienced on each handoff. Each of the four types of handoffs were performed (randomly) 84 times; the frequencies are based on the aggregate total set of handoffs. The handoffs cluster as we expected, with the laptop-to-laptop values showing the lowest latency ($\mu = 1.2s, \sigma = 0.26s$); the pda-to-laptop values are next quickest, with $\mu = 3.9s, \sigma = 0.20s$; the laptop-to-pda values ($\mu = 9.0s, \sigma = 0.17s$) and pda-to-pda results ($\mu = 10.6s, \sigma = 0.17s$) show a distinctly longer latency.

This increased latency is dominated by the time it takes to load and begin execution of the JTA application on the PDA: while the startup cost on the laptop averages 0.48s, on the PDA the average is 6.8s. If the PDA's application startup cost was more like the laptop, the user-visible delay would be quite comparable. Recall that this application was not "ported" to the PDA; the code was simply loaded into its persistent storage and executed. We can reasonably conjecture that a version of JTA tuned for a PDA-class device would be much leaner and exhibit a strikingly lower startup delay, and we therefore conclude that heterogeneous application session handoff of complex applications with extensive state and a large code base is achievable while incurring modest delay.

4.4 Video player experiment

The previous experiments explored application session handoff in the context of discrete data. This experiment examines how well iMASH performs on streaming data, since latency is critical to the user experience of real-time streams. The overall system architecture (application server, middleware server, client) is standard iMASH. We developed a simple streaming server and matching display application based on Sun's Java Media Framework (JMF) [Sunb], and also incorporated JMF into the content adaptation pipeline (CAP) component of an RPC-based iMASH middleware server. JMF had the benefits of being Java-based—as is our MWS—and it has

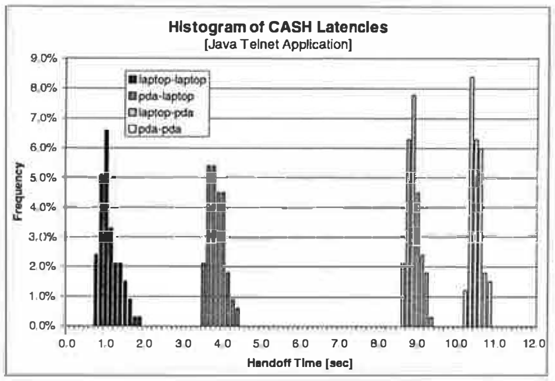


Figure 8: Frequency histogram of client latency experienced on client-only handoff (i.e., single MWS), for ≈ 340 handoffs of JTA application.

an extensive library of format transcoders suitable for a range of output stream bandwidths. It also uses the Real-Time Transport Protocol (RTP) [IET] as a delivery substrate. A downside is that JMF is not designed to switch transcoding methods on-the-fly, and so the CAP implementation must destroy and reconstruct a JMF instance to change the transcoding in mid-stream.

In our first streaming data experiment, we assessed the basic cost of switching transcoding methods, in the absence of handoff. In this experiment, an external

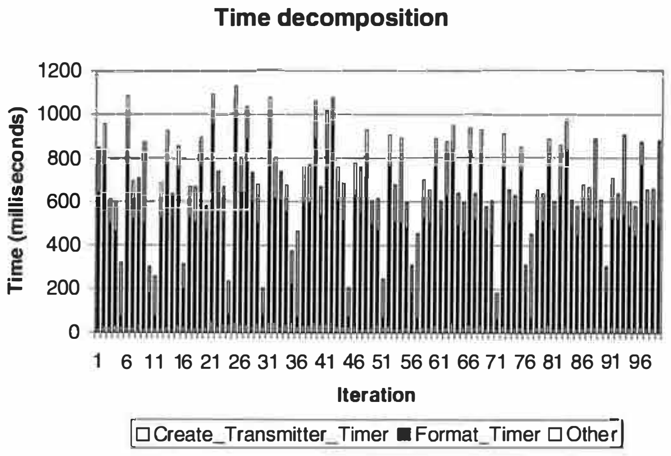


Figure 9: Latency to switch video transcoding method

thread notified the CAP that a significant change in available MWS-to-client bandwidth had occurred, and a new transcoding should be considered.[11] The CAP selects an appropriate transcoding method, kills the current JMF, and restarts a new one. The graph in Figure 9 shows the latencies experienced for a series of about 100 artificially-induced transcoding format changes. The alpha (=0.05) mean is 694ms to switch transcodings, but substantial unexpected variance clearly is visible in both the overall values (the entire vertical bar) and especially in the "Format Timer" component—typically 2/3 of the total.

Closer study revealed that this component delay is highly correlated with the transcoder selected: all of the five transcoders used in the experiment showed much less variance when grouped by transcoder type–but ranged from a mean of 50ms for the cheapest method to a mean of about 500ms for the most expensive. Sun's documentation for this portion of JMF (see [Suna]) indicates that this step may be very time consuming, as JMF may need to communicate with a server, read from a file, etc., in the course of obtaining its required resources. The choice of transcoder type (and perhaps implementation) is a dominant factor in our results, with "nicer" delays around 200ms readily achievable.

Our second streaming data experiment measured the cost to perform a CASH (client-only application session handoff) on the video player application used above. In this experiment, a user begins to play a video stream, and then randomly performs a series of handoffs back and forth between two clients. We measured the length of the interval between the time when the MWS received a message from the source client requesting handoff, and the time when the MWS began producing a stream for the target client. Note that with such applications, the latency experienced by the user is heavily dependent on size and content of the player's input buffer, which is frequently many seconds worth of data.[LGGB02].

The results of about 80 handoffs are shown in Figure 10. The mean delay is 1567ms, with a standard devi-

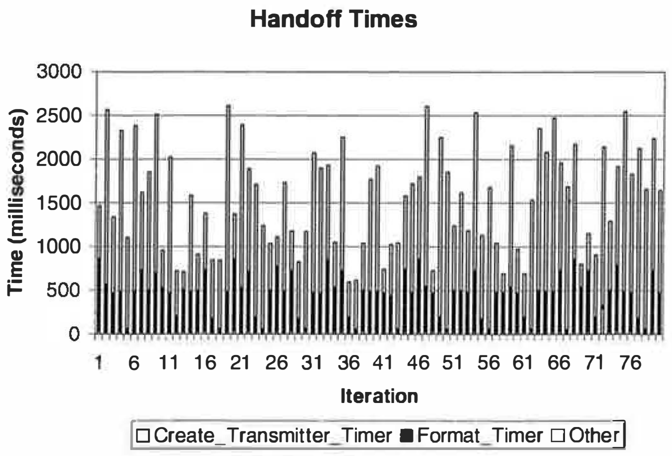


Figure 10: Streaming video handoff latency

ation of 587ms. Here again we see a noticeable variance, much of which is attributable to the cost to create a new transcoder object within JMF. Since the typical delay is well under two seconds, we conclude from this experiment that very fast heterogeneous application session handoff

[11]Effective ways to learn at the middleware level of bandwidth and other network QoS changes is the subject of ongoing research in our lab.

of a streaming data application is feasible using iMASH.

5 Conclusions and Future Work

Mobile continuous computing in the medical domain has been challenged to date by a lack of software support for the inherent heterogeneity found there. iMASH fills the gap in a significant way by exploiting *application session handoff*, a novel legacy server unaware, client application aware, secure approach to moving a computation across heterogeneous platforms with very low latency.

The architecture presented in this paper has been validated by experience with a number of prototype legacy applications and middleware server implementations. Further proof of the iMASH concept is found in experimental data obtained from implementations, which shows that the user-visible latency incurred by application session handoff is sufficiently low (typically, under two seconds) to allow deployment of iMASH in the real world—and enable the opening scenario to reach fruition.

The iMASH architecture supports streaming data, and our "streaming CAP" rapid prototype is currently being re-engineered for integration into our production middleware implementation. Construction of a detailed simulation model of iMASH is also underway. We expect the model to validate early design decisions which had foreseeable scalability implications, and also anticipate insight on unforeseen scalability opportunities.

Future work will look at several critical areas. It is important that a deeper understanding of interaction between layers be obtained, especially between wireless (re)transmission and content adaptation at higher layers. We also need to better understand (and perhaps loosen) the layer boundaries between mobile IP and ASH: cross-layer interaction may well be appropriate here.

The iMASH architecture, with multiple middleware servers and multiple client devices, appears ripe for enhanced reliability and robustness. Extensive caching at both MWS and client, coupled with mobility-inspired handoff, should be exploitable for automatic fail-over in response to either MWS or client failure.

Finally, client devices sometimes act as significant data sources (essentially servers), such as when participating in a video conference. The extent to which the iMASH architecture can be inverted is worthy of further study.

References

[BGW01] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting mobile communications: the insecurity of 802.11. In *Proceedings of the MOBICOM*, 2001.

[FGBA96] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the 7th ASPLOS*. ACM, October 1996.

[GDLB02] Robert Grimm, Janet Davis, Eric Lemar, and Brian Bershad. Migration for pervasive applications. Submitted for publication, 2002.

[Hel00] David Hellerstein. HIPAA and health information privacy rules: Almost there. *Health Mgt. Tech.*, April 2000.

[IET] IETF. RTP: Real time protocol, RFC3267. ftp://ftp.rfc-editor.org/in-notes/rfc3267.txt.

[JM] Matthias Jugel and Marcus Meissner. JTA, java telnet/SSH application. http://www.javassh.org.

[LGGB02] Jinsong Lin, Glenn Glazer, Richard Guy, and Rajive Bagrodia. Fast asynchronous streaming handoff. In *Proceedings of the IDMS/PROMS 2002*, 2002.

[LL02a] Wai Lum and Francis Lau. On balancing between transcoding overhead and spatial consumption in content adaptation. In *MOBICOM proceedings*, 2002.

[LL02b] Wai Yip Lum and Francis C.M. Lau. A context-aware decision engine for content adaptation. *IEEE Pervasive Computing*, pages 41–49, July 2002.

[MD00] D. Milojicic and F. Douglis, et al. Process migration survey. In *ACM Computing Surveys*, 2000.

[Nob97] Brian D. Noble, et al. Agile application-aware adaptation for mobility. In *Proceedings of the 16thSOSP*, pages 276–287. ACM, October 1997.

[PGB01] Thomas Phan, Richard Guy, and Rajive Bagrodia. A scalable, distributed middleware service architecture to support mobile internet applications. In *Proceedings of the IEEE Workshop on Wireless Mobile Internet*, 2001.

[PZB02] Thomas Phan, George Zorpas, and Rajive Bagrodia. An extensible and scalable content adaptation pipeline architecture to support heterogeneous clients. In *Proceedings of the 22nd ICDCS*, 2002.

[RKC01] Manuel Román, Fabio Kon, and Roy H. Campbell. Reflective middleware: From your desk to your hand. *Distributed Systems Online*, 2(5), 2001. http://dsonline.computer.org.

[SB00] Alex C. Snoeren and Hari Balakrishnan. An end-to-end approach to host mobility. In *ACM/IEEE Int'l. Conf. on Mobile Computing and Networking*, August 2000.

[SKP02] Erik Skow, Jiejun Kong, and Thomas Phan, et al. A security architecture for application session handoff. In *Proceedings of the ICC*, 2002.

[Suna] Sun Microsystems. javax.media interface controller. http://java.sun.com/products/java-media/jmf/2.1.1/apidocs/javax/media/Controller.html.

[Sunb] Sun Microsystems. JMF: Java media framework. http://java.sun.com/products/java-media/jmf/.

[vMCT95] Erik van Mulligen, Ronald Cornet, and Teun Timmers. Problems with integrating legacy systems. In *Annual Symposium on Computer Applications in Medical Care : Toward Cost-Effective Clinical Computing*, 1995.

[WAP] WAP Forum. Wireless transport layer security specification. www1.wapforum.org/tech/ documents/WAP-261-WTLS-20010406-a.pdf.

[ZM02] Victor C. Zandy and Barton P. Miller. Reliable network connections. In *ACM/IEEE Int'l. Conf. on Mobile Computing and Networking*, 2002.

Tactics-Based Remote Execution for Mobile Computing

Rajesh Krishna Balan[†], Mahadev Satyanarayanan[†‡], SoYoung Park[†], Tadashi Okoshi[†]
[†]Carnegie Mellon University and [‡]Intel Research Pittsburgh
{rajesh,satya,seraphin,slash}@cs.cmu.edu

Abstract

Remote execution can transform the puniest mobile device into a computing giant able to run resource-intensive applications such as natural language translation, speech recognition, face recognition, and augmented reality. However, easily partitioning these applications for remote execution while retaining application-specific information has proven to be a difficult challenge. In this paper, we show that automated dynamic re-partitioning of mobile applications can be reconciled with the need to exploit application-specific knowledge. We show that the useful knowledge about an application relevant to remote execution can be captured in a compact declarative form called **tactics**. *Tactics capture the full range of meaningful partitions of an application and are very small relative to code size. We present the design of a tactics-based remote execution system, Chroma, that performs comparably to a runtime system that makes perfect partitioning decisions. Furthermore, we show that Chroma can automatically use extra resources in an over-provisioned environment to improve application performance.*

1 Introduction

Remote execution can transform the puniest mobile device into a computing giant. This would enable resource-intensive applications such as natural language translation, speech recognition, face recognition, and augmented reality to be run on tiny handheld, wearable or body-implanted platforms. Nearby compute servers, connected through a low-latency wireless LAN, can provide the CPU cycles, memory, and energy needed for such applications.

Unfortunately, two annoying facts cloud this rosy future. First, the optimal partitioning of an interactive application into local and remote components is highly application-specific and platform-specific. Since mobile hardware evolves rapidly, this optimal partitioning changes on the timescale of months rather than years. Suboptimal partitioning can result in sluggish and intolerable interactive response. Hence, a tight and ongoing coupling between application developers and hardware platform developers appears inevitable. Second, matters are made worse by the fact that mobile environments exhibit highly variable resource availability. Bandwidth, energy and presence of compute servers can change on the timescale of minutes or hours, as a user moves to different locations. Re-partitioning an application for changed operating conditions at this timescale is therefore essential. These considerations suggest that an automated approach to partitioning applications for remote execution is necessary. However, partitioning an application without taking into consideration its unique characteristics may result in sub-optimal partitions.

Can automated dynamic re-partitioning be reconciled with the need to exploit application-specific knowledge? In this paper, we show that this is indeed possible. Our key insight is that *the knowledge about an application relevant to remote execution can be captured in compact declarative form that is very small relative to code size.* More specifically, the full range of meaningful partitions of an application can be described in a compact external description called *remote execution tactics* or just *tactics* for brevity. Thus, the tactics for an application constitute the limited and controlled exposure of application-specific knowledge necessary for making effective partitioning and placement decisions for that application in a mobile computing environment.

In this paper, we examine three applications of the genre mentioned earlier (natural language translation, speech recognition, and face recognition) and show that the tactics for each is much less than one percent of total code size. We present the design of *Chroma*, a tactics-based remote execution system, and show that sound partitioning and placement of these applications using tactics is possible. We show that Chroma is able to achieve application performance that is comparable to execution on an ideal runtime system.

In addition, we show that Chroma can opportunistically utilize extra resources in an over-provisioned environment. This allows us to achieve lower latencies for the three applications mentioned above.

The rest of this paper is organized as follows: Section 2 presents the assumptions and goals of this work while Section 3 presents the design of Chroma. We present our experimental setup in Section 4. Sections 5 and 6 present Chroma's performance relative to an ideal runtime system. In Section 7, we show how tactics can improve application performance in the presence of extra resources. Section 8 presents related work and Section 9 concludes the paper.

2 Design Rationale

2.1 Assumptions

In this work, we assume that all code necessary for remote execution is already present on all the clients and servers. We do not perform any code migration and use coarse-grained remote execution on the order of seconds. This granularity is appropriate for the class of applications being targeted. This is in contrast to other remote execution systems, like Java RMI [21], that perform fine-grained remote execution on the order of microseconds. We assume that the individual remote calls that make up the remote execution are self-contained and do not produce side effects.

Since Chroma is meant to be used on mobile devices, we assume a highly variable resource environment. Network characteristics and remote infrastructure available for hosting computation vary with location. File cache state and CPU load on local and remote machines significantly impact application performance. Application energy consumption varies depending upon the specific platform on which an application executes. Variation in any resource can significantly change the best placement of functionality. Thus, Chroma must continually monitor resource availability and adapt to changes in the environment.

The class of applications that we are targeting are computationally intensive interactive applications. Examples include speech recognition, natural language translation and augmented reality applications. These are the kinds of applications that have been envisioned as being key mobile applications in the near future [18, 23].

We assume that Chroma will not require applications to be developed from scratch. Instead Chroma will use existing applications that have been slightly modified to work with Chroma. This is a realistic assumption because building new applications from scratch requires huge amounts of effort. This is likely to be unprofitable when application development time becomes comparable to the useful lifetime of the wearable and/or handheld hardware being targeted. In this paper, we do not address the security and admission control issues involved in using remote servers.

2.2 Goals

Chroma was designed to achieve three major goals. These are:

- *Seamless from user perspective*: The user should be oblivious to the decisions being made by Chroma and the actual execution of those decisions.
- *Effectiveness* : Chroma should employ close to optimal strategies for remote execution under all resource conditions. An application developer should not be tempted to hand tune.
- *Minimal burden on application writers*: We want Chroma to be an easy system for application writers to use.

2.3 Solution Strategy

2.3.1 Seamless from user perspective

We achieved this goal by making Chroma completely automatic from the perspective of the application user. Chroma was designed to work with interactive applications which demand user attention due to their interactive nature. As such, Chroma was designed to require minimal additional user attention. The user specifies high-level preferences in advance to Chroma to guide its decision making process. With these preferences, Chroma will decide at runtime how and where to execute applications. The user is oblivious to these decisions in normal use of the system.

2.3.2 Effectiveness

To achieve the best possible performance, Chroma should use the optimal strategy for remote execution for any particular resource condition. But how do we determine what that optimal strategy is? In theory, it is possible, for every resource condition, to test every single way of splitting an application for remote execution and then picking the best strategy. However, this is intractable in practice. Another method is just to pick one possible way of splitting up the application and using it all the time. However, this static method will be ineffective when resources change. The key insight that allows us to achieve our performance goal while keeping the search space small is this:

For every application, the number of useful ways of splitting the application for remote execution is small.

We call these useful ways of splitting the application the *tactics* of the application. Tactics are specified by the application developer and are high level descriptions of meaningful module-level partitions of an application. Our experience with modifying applications in the course of this work suggests that it is easy for an application developer to provide the tactics for an application.

An application is made up of *operations*. An operation is an application-specific notion of work. Tactics enumerate the various ways that an operation can be usefully executed. For example, an operation for a speech recognition application would be `recognize_utterance` while an operation for a graphics application would be

render. For each operation, the application developer specifies one or more tactics. These different tactics may differ in the amount of resources they use and their *fidelity* [15]. Fidelity refers to an application specific metric of quality. For example, speech recognition has higher fidelity when using a large vocabulary rather than a small vocabulary. Fidelity ranges from 0 to 1, with 1 being the best quality and 0 the worst.

2.3.3 Minimal Burden on Application Writers

Developing mobile computing applications is especially difficult because they have to be adaptive [6, 10]. The resource constraints of mobile devices, the uncertainty of wireless communication quality, the concern for battery life, and the lowered trust typical of mobile environments all combine to complicate the design of mobile applications. Only through dynamic adaptation in response to varying runtime conditions can applications provide a satisfactory user experience. Unfortunately, the complexity of writing and debugging adaptive code adds to the application software development time. Hence, instead of building the mechanisms to detect resource availability and trigger adaptation directly into each application, we created a runtime system that provides this functionality. However, the question still remains: How do we easily modify existing applications to use the adaptation features provided by our runtime?

Our approach to achieving this goal can be summarized as follows: First, we provide a lightweight semi-automatic process for customizing the API used by the application. Such customization is targeted to the specific needs of the application. Second, we provide tools for automatic generation of code stubs that map the customized API to the generic API used by Chroma. Finally, this generic API is supported by Chroma, which monitors resource levels and triggers application adaptation. Chroma support also helps ensure that the adaptations of multiple concurrently executing applications do not interfere with each other. Further details about the software engineering aspects of Chroma can be found elsewhere [2].

3 Chroma Design

In this section, we present the design of our tactics-based remote execution system, Chroma, that satisfies the goals described in Section 2. Building Chroma required two main components:

- A way of describing tactics.
- A method for selecting a tactic at runtime.

3.1 Describing Tactics

Figure 1 shows the tactics for Pangloss-Lite, a natural language translator. Pangloss-Lite uses up to three translation engines (*dictionary*, *ebmt* and *glossary*) to translate a sentence. The tactics specify the different ways of combining these engines and are composed of two distinct portions. Using more than one engine results in a better translation but at the cost of using more resources.

The first portion of the description (denoted by the keyword *RPC*) details the remote calls that can be used for this application. The second portion (denoted by the keyword *DEFINE_TACTIC*) defines the specific sequence of remote calls that make up a particular tactic. An "&" separator between remote calls denotes that the remote calls must be performed in sequential order while remote calls within brackets ((**server_gloss**, **server_ebmt**)) tell the remote execution system that those calls can be executed in parallel.

Each tactic fully describes one way of combining RPCs to complete an operation. The data dependencies between RPCs are visible because the prototypes of the remote calls are specified in the tactics description. Each of the individual remote calls that make up a particular tactic can be run either locally or on any remote server. This decision is made at runtime. Even though the tactics may differ in their resource usage and fidelity, each tactic is guaranteed to produce a proper result for the given operation if the remote calls are performed in the order specified by the tactic (we assume no side effects as mentioned in Section 2.1). Since the data dependencies and ordering between remote calls is fully specified by the tactic description, Chroma is able to parallelize the execution of these remote stages whenever possible. This aspect of Chroma is explained further in Section 3.3.

A key point to note is that the description of the application's tactics is very small compared to the size of the application. As shown in Figure 1, it requires about 14 lines to specify the tactics for Pangloss-Lite. This is in comparison to the roughly 150K lines of code in Pangloss-Lite.

3.2 Tactic Selection

In this section we highlight the system components necessary for Chroma to decide at runtime which tactic to use and where to execute it. For example, if Chroma picks the tactic **gloss_ebmt** (Figure 1) for Pangloss-Lite, it will also have to decide whether to execute the **server_gloss**, **server_ebmt** and **server_lm** remote calls of this tactic locally or remotely. Chroma's goal is thus to decide on a *tactic plan*. A tactic plan is comprised of a tactic number (denoting which tactic to use) along

```
RPC server_gloss (IN string line, OUT string gloss_out);
RPC server_dict  (IN string line, OUT string dict_out);
RPC server_ebmt  (IN string line, OUT string ebmt_out);
RPC server_lm    (IN string line, IN string ebmt_out,
                  IN string dict_out, IN string gloss_out,
                  OUT string translation);

DEFINE_TACTIC gloss   =       server_gloss & server_lm;
DEFINE_TACTIC dict    =       server_dict & server_lm;
DEFINE_TACTIC ebmt    =       server_ebmt & server_lm;
DEFINE_TACTIC gloss_dict =    (server_gloss, server_dict) & server_lm;
DEFINE_TACTIC gloss_ebmt =    (server_gloss, server_ebmt) & server_lm;
DEFINE_TACTIC dict_ebmt  =    (server_dict, server_ebmt) & server_lm;
DEFINE_TACTIC gloss_dict_ebmt = (server_gloss, server_dict, server_ebmt) & server_lm;
```

Pangloss-Lite has seven tactics that are listed after the DEFINE_TACTIC keyword. These seven tactics give different ways of combining the remote calls (listed after the keyword RPC) for this application. Each of these calls can be executed locally or at a remote server and this is determined at runtime by Chroma

Figure 1: Tactics for Pangloss-Lite

with a list that specifies the server to use for each RPC in that tactic. Using the local machine avoids network transmission and is unavoidable if the client is disconnected. In contrast, using a remote machine incurs the delay and energy cost of network communication but exploits the CPU and energy resources of a remote server. Chroma enumerates through all possible tactic plans and picks the best one for the given resource availability.

To be able to do this, first, Chroma needs to be able to predict the resource usage of each tactic plan. Second, Chroma has to measure the current resource availability. Third, Chroma requires guidance from the user about the relative importance of each resource. Given these three things, Chroma will be able to decide on the best tactic.

3.2.1 Resource Prediction

For a given operation and tactic plan, Chroma needs to be able to predict the resources the tactic plan will require. This information is provided by resource demand predictors that use history based prediction [14]. The key idea here is that the resource usage of a tactic plan can be predicted from its recent resource usage. The demand prediction mechanisms are initialized by off-line logging. At runtime, these predictors are updated using online monitoring and machine learning to improve accuracy.

3.2.2 Resource Monitoring

Chroma uses multiple resource measurers to determine current resource availability. These resource measurers currently measure memory usage, CPU availability, available bandwidth, latency of operation, file cache state and battery energy remaining. Chroma also has mechanisms to retrieve resource availability information from remote servers.

3.2.3 User Guidance

To effectively match resource demand to resource availability, Chroma needs to trade off resources for fidelity. How to perform this tradeoff is frequently context sensitive and thus dynamic. For instance, would the user of a language translator prefer accurate translations or snappy response times? Should an application running on a mobile device use power-saving modes to preserve battery charge, or should it use resources liberally in order to complete the user's task before he or she runs off to board their plane? That knowledge is very hard to obtain at the application level as it is user-specific and not application-specific.

We provide Chroma with these user-specific resource tradeoffs in the form of *utility functions*. A utility function is a user-specific function that quantifies the tradeoff between two or more attributes.

In this paper, we use a fixed utility function that states that latency is as important as fidelity and that Chroma should ignore battery lifetimes. Chroma will thus choose the tactic plan that maximizes the *latency-fidelity metric* (expressed mathematically as maximizing the quantity $\frac{fidelity}{latency}$). In our future work, we plan to develop methods that will allow us to capture different utility functions from the user using a graphical user interface. These different utility functions will allow us to optimize the tactic selection for other user specified metrics like conserving battery power or minimizing network bandwidth.

3.2.4 Selection Process

Figure 2 shows how all the components work together. Chroma determines expected resource demand for each

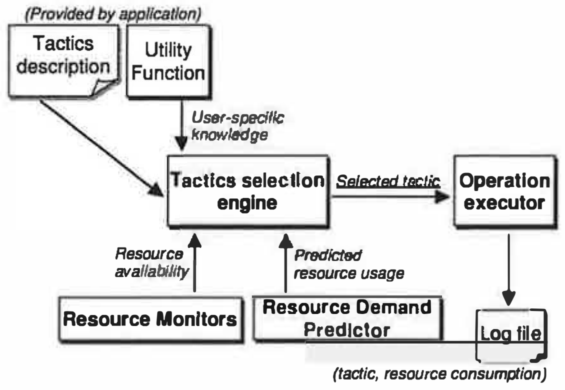
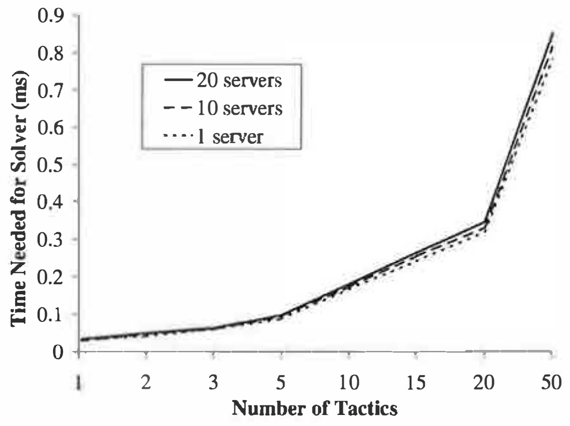


Figure 2: Choosing a Tactic



This figure shows the overhead incurred (in milliseconds) by the solver in deciding which tactic to choose. The overhead shown is only for the computational aspect of the solver and does not include the time needed by other parts of Chroma such as the resource estimators and resource demand predictors. To obtain these results, we extracted the core solver from Chroma and supplied it with synthetic inputs. This allowed us to measure just the overhead of the solver. The total measured overhead of Chroma is shown in Section 6.

Figure 3: Overhead of Choosing a Tactic

tactic of the current operation by querying the resource prediction component. At the same time, Chroma determines the available resources via the resource monitoring component. These resource monitors also query any available remote servers to determine the resource availability on those servers. This information is necessary as the latency of the tactic is determined by where each individual remote call in that tactic is being executed. Determining resource availability on demand can be a very time consuming operation. Hence, to improve performance at the cost of accuracy, the resource monitors perform these queries periodically in the background and cache the results.

Chroma iterates through every possible tactic plan and picks the best tactic plan to use for this operation. It does this by picking the tactic plan that maximizes the latency-fidelity utility function metric. The tactic plan is then executed and its resource usage is logged to refine future demand prediction. This brute force method works well for a small number of tactics as shown in Figure 3. From the results in Section 6, we see that the contribution of the solver to the total Chroma overhead is minimal. We claim that, in practice, the number of useful tactics for computationally intensive interactive applications is small enough to allow this brute force tactic selection mechanism. We are currently verifying this claim and also looking at using other solvers that are both less computationally demanding and provably correct [11].

3.3 Over-Provisioned Environments

Our discussion so far has focused on environments that are resource constrained. However, environments such as smart rooms, may be *over-provisioned.* Over-provisioned environments are characterized as having more computing resources than are actually needed for normal operation. We would like to have a system that works well if resources are scarce but is able to immediately make use of over-provisioning if it becomes available. Our goal is to exploit idle resources to improve user experience.

Tactics help by providing the knowledge of the remotes calls needed by a given operation and the data dependencies between them. Chroma can use the knowledge in tactics opportunistically to improve user experience in three different ways.

First, Chroma can make multiple remote execution calls (for the same operation) to remote servers and use the fastest result. For example, Chroma can execute the glossary engine of Pangloss-Lite at multiple servers and use the fastest result. Chroma knows that it can do this safely because the description of the tactics makes it clear that executing the glossary engine is a stand-alone operation and does not require any previous results or state. We call this optimization method "fastest result".

Second, Chroma can split the work necessary for an operation among multiple servers. It does this by decomposing operation data into smaller chunks and shipping each chunk to a different remote server. Chroma uses hints from the application to determine the proper method of splitting operation data into smaller chunks. We call this optimization "data decomposition".

Third, Chroma can perform the same operation but with different fidelities at different servers. Chroma can then return the highest fidelity result that satisfies the latency constraints of the application. For example, Chroma can execute multiple instances of the `ebmt` engine of Pangloss-Lite in parallel at separate servers (all with different fidelities) and use the highest fidelity result that has returned before a specified amount of time. We call this optimization method "best fidelity".

Tactics allow us to use these optimizations on behalf of applications automatically without the applications needing to be re-compiled or modified in any way. There are other optimizations possible with tactics, but these are the ones we have explored so far and we present performance results for them in Section 7.

4 Validation Approach

4.1 Applications

To validate the design of Chroma, we have used three applications that are representative of the needs of a future mobile user. These applications are all computationally intensive interactive applications that are currently being actively developed for mobile environments. These applications are

- Pangloss-Lite [7] : A natural language translator written in C++ for translating sentences in one language to another. This kind of application is important for the modern mobile user who is moving from country to country.
- Janus [22] : A speech to text conversion program written in C that can be used to convert voice input into text. This kind of application is at the core of any voice recognition system that is used to control mobile devices.
- Face [20] : A program written in Ada that detects faces in images and is representative of image processing applications. Surveillance personnel, with wearable computers, that use images to detect suspicious features in the environment are likely to require this kind of application.

4.2 Experimental Platform

We used HP Omnibook 6000 notebooks with 256 MB of memory, a 20 GB hard disk and a 1 GHz Mobile Pentium 3 processor as our remote servers.

We used two different clients that represent the range of computational power available in today's mobile devices. The *fast client* is the above mentioned HP Omnibook 6000 notebook. The *slow client* is an IBM Thinkpad 560X notebook with 96 MB of memory and a 233 MHz Mobile Pentium MMX CPU. The computational power of the Thinkpad 560X is representative of today's most powerful handheld devices.

The clients and servers ran Linux and were connected via a 100 Mb/s Ethernet network. A deployed version of Chroma would use a wireless LAN such as 802.11a (55 Mb/s). We used the Coda [19] distributed file system to share application code between the clients and servers.

4.3 Success Criteria

To successfully validate Chroma, we need to show the following things:

- Chroma is able to correctly pick the best tactic plan for a particular application and resource availability. We demonstrate this by showing that Chroma picks the tactic plan that maximizes (or comes close to maximizing) the latency-fidelity metric.
- The overhead of Chroma's decision making process is not too large and does not add substantially to the total latency of the application.
- Chroma is able to use tactics to automatically improve application performance in the presence of additional server resources.

The validation of these three parts will justify our claim that tactics are a valuable. Sections 5, 6 and 7 present our results relative to the above points.

5 Results: Tactic Selection

Since Chroma automatically determines how to remotely execute an application based on the current resources, it is possible that the decisions it makes are not as good as a careful manual remote partitioning of the application. We allay this concern by showing that Chroma's partitioning comes close to the optimal partitioning possible for a number of different applications and operating conditions.

To demonstrate this, we compare the decision making of Chroma with that of an ideal runtime system. This ideal runtime system is achieved by manually testing every possible tactic plan for a given experiment and then choosing the best one. Chroma, on the other hand, has to figure out the best tactic plan dynamically at runtime. We define the best tactic plan as being the one that maximizes the latency-fidelity metric. We show that Chroma chooses a tactic plan that either maximizes the latency-fidelity metric or comes very close to it.

Each experiment was repeated five times and our results are shown with 90% confidence intervals where applicable. Since Chroma uses history-based demand prediction, we created history logs for each application before running the experiments using training data that was not used in the actual experiments. These logs provide the system with the proper prediction values for the application. Without these logs, the system would have to slowly learn the correct prediction values online and this could take a long time.

Sentence Length (No. of Words)	Ideal Runtime		Chroma		Ratio
	chosen tactic	metric	chosen tactic	metric	
11	gloss_dict_ebmt	1.00	gloss_dict_ebmt	1.00	1.00
23	gloss_dict_ebmt	1.00	dict_ebmt	0.70	0.70
35	gloss_dict_ebmt	1.00	dict_ebmt	0.70	0.70
47	gloss_dict_ebmt	0.70	dict_ebmt	0.70	1.00
59	dict_ebmt	0.70	dict_ebmt	0.70	1.00

(a) Fast Client

Sentence Length (No. of Words)	Ideal Runtime		Chroma		Ratio
	chosen tactic	metric	chosen tactic	metric	
11	gloss_dict_ebmt	1.00	gloss_dict_ebmt	1.00	1.00
23	gloss_dict_ebmt	1.00	gloss_dict_ebmt	1.00	1.00
35	gloss_dict_ebmt	1.00	dict_ebmt	0.70	0.70
47	dict_ebmt	0.70	dict_ebmt	0.70	1.00
59	dict_ebmt	0.70	dict_ebmt	0.70	1.00

(b) Slow Client

This table shows the tactic plan chosen by Chroma and the ideal runtime. The locations chosen by Chroma and the ideal runtime were identical in all cases and are thus omitted from the table. We also show the value of the latency-fidelity metric for the tactic plans chosen by the two systems. The ratio ($\frac{Chroma}{Ideal}$) between the ideal system's metric and Chroma's is shown in the Ratio column.

Figure 4: Comparison Between the Ideal Runtime and Chroma for Pangloss-Lite

5.1 Pangloss-Lite

5.1.1 Description

As mentioned in Section 3.1, Pangloss-Lite translates text from one language to another. It can use up to three translation engines: EBMT (example-based machine translation), glossary-based, and dictionary-based. Each engine returns a set of potential translations for phrases within the input text. A language modeler combines their output to generate the final translation.

Pangloss-Lite's fidelity increases with the number of engines used for translation. We assign the EBMT engine a fidelity of 0.5. The glossary and dictionary engines produce subjectively worse translations—we assign them fidelity levels of 0.3 and 0.2, respectively. When multiple engines are used, we add their individual fidelities since the language modeler can combine their outputs to produce a better translation. For example, when the EBMT and glossary-based engines are used, we assign a fidelity of 0.8. The seven possible combinations of the engines are captured by the seven tactics (shown in Fig 1).

We use the latency-fidelity utility function to determine the tactic to use for Pangloss-Lite. However, to model the preferences of an interactive user, we specify that all latencies of one second or lower are equally good and that all latencies larger than five seconds are impossibly bad. Thus if the latency is greater than five seconds, we set the latency to a really large number (thus making the utility value really small) and if the latency is one second or lower, we set the latency value to one. All other latency values are left unchanged.

All three engines and the language modeler may be executed remotely. While execution of each engine is optional, the language modeler must always execute. Thus, there are at least 52 tactic plans from which Chroma may choose when at least one remote server is available.

We used as input five sentences with different number of words (ranging from 11 words to 59 words) as inputs for the baseline experiments. The input sentences were in Spanish and were translated into English. There were three remote servers available and both the servers and the clients were unloaded for the purposes of this experiment.

5.1.2 Results

Figures 4 displays the decisions made by Chroma compared with the decisions made by the ideal runtime for each sentence on the fast and slow clients respectively. From the results, we see that Chroma made decisions that approximated the decisions made by the ideal runtime system. In the cases where Chroma made a different decision, it was off by 30%. This difference in decision making was due to incorrect resource estimations by Chroma. From the results, we see that Chroma decided not to run the `glossary` engine in the cases where it differed from the ideal runtime. The time needed for the glossary engine to complete a translation was hard

Utterance	Ideal Runtime		Chroma		Ratio
	chosen tactic	metric	chosen tactic	metric	
1	reduced	0.50	reduced	0.50	1.00
2	reduced	0.50	reduced	0.50	1.00
3	reduced	0.50	reduced	0.50	1.00
4	reduced	0.50	reduced	0.50	1.00
5	reduced	0.50	reduced	0.50	1.00
6	reduced	0.50	reduced	0.50	1.00
7	full	0.53	reduced	0.50	0.94
8	reduced	0.50	reduced	0.50	1.00
9	reduced	0.50	reduced	0.50	1.00
10	reduced	0.50	reduced	0.50	1.00

(a) Fast client

Utterance	Ideal Runtime		Chroma		Ratio
	chosen tactic	metric	chosen tactic	metric	
1	reduced	0.50	reduced	0.50	1.00
2	reduced	0.50	reduced	0.48	0.96
3	reduced	0.50	reduced	0.50	1.00
4	reduced	0.50	reduced	0.50	1.00
5	reduced	0.50	reduced	0.49	0.98
6	reduced	0.50	reduced	0.50	1.00
7	reduced	0.50	reduced	0.50	1.00
8	reduced	0.42	reduced	0.41	0.98
9	reduced	0.50	reduced	0.50	1.00
10	reduced	0.50	reduced	0.48	0.96

(b) Slow Client

This table shows the tactic plan chosen by Chroma and the ideal runtime. The locations chosen by Chroma and the ideal runtime were identical in all cases and are thus omitted from the table. We also show the value of the latency-fidelity metric for the tactic plans chosen by the two systems. The ratio ($\frac{Chroma}{Ideal}$) between the ideal system's metric and Chroma's is shown in the Ratio column.

Figure 5: Comparison Between the Ideal Runtime and Chroma for Janus

```
RPC do_full_recognition
               (IN string utterance,
               OUT string translation);

RPC do_reduced_recognition
               (IN string utterance,
               OUT string translation);

DEFINE_TACTIC full_recognition =
                do_full_recognition;

DEFINE_TACTIC reduced_recognition =
                do_reduced_recognition;
```

The tactics declaration for Janus contains two remote calls (do_full_recognition and do_reduced_recognition) that can be run either locally or remotely.

Figure 6: Tactics for Janus

for Chroma to predict as it was not a simple function of the length of the input sentence. Chroma's decision to drop the `glossary` engine incurred a 30% reduction in fidelity and this resulted in the final 30% difference in Ratio. The latencies used to calculate the metric were below 1 second for both Chroma and the ideal runtime system in all the cases where the metrics differed.

5.2 Janus

5.2.1 Description

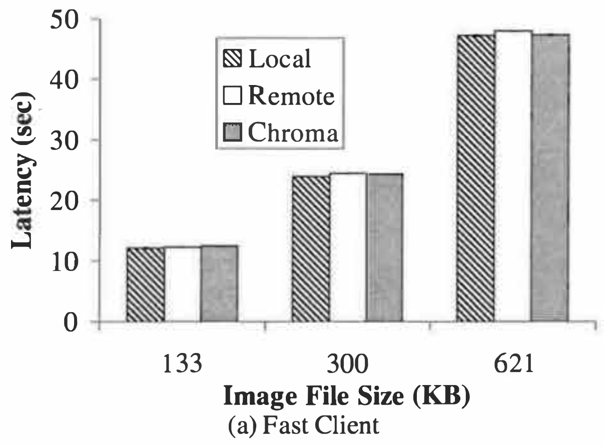
Janus performs speech-to-text translation of spoken phrases. Recognition can be performed at either full or reduced fidelity. The reduced fidelity uses a smaller, more task-specific vocabulary that limits the number of phrases that can be successfully recognized but requires less time to recognize a phrase. We assign the reduced fidelity a utility of 0.5 and the full fidelity a utility of 1.0 to reflect this behavior. Similar to Pangloss-Lite, we model an interactive user by making all latencies less than or equal to one second equally good (we set the latency value to one) and all latencies greater than five seconds horribly bad (we set the latency to a really large number). All other latency values are left unchanged.

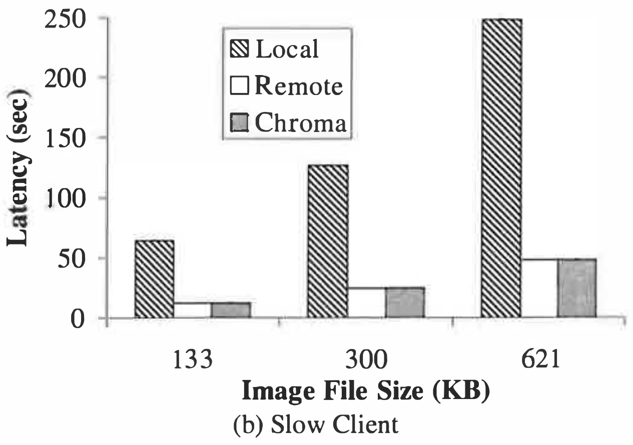
Janus has two remote calls that can be executed either locally or remotely. These two possible ways of executing Janus are captured by Janus's tactics, as shown in Figure 6. The tactic `full_recognition` uses the full fidelity vocabulary to do the recognition while the tactic `reduced_recognition` uses the reduced fidelity vocabulary to do the recognition. Describing Janus's tactics requires 4 lines of code in our declarative language. This is significantly smaller than Janus itself which is ≈120K lines of C code.

We used as input ten different utterances containing different numbers of spoken words (ranging from 3 words to 10 words) as inputs for the baseline experiments. One remote server was used for this experiment and both the server and the clients were unloaded.

5.2.2 Results

Figure 5 shows the decisions made by the ideal runtime and Chroma. We see that Chroma picked the optimal choice in almost all cases on the fast client. Even in the case where Chroma picked a different tactic plan, the latency-fidelity metric of the plan picked by Chroma was very close to optimal (94% of optimal). On the slow client, Chroma performed as well as the ideal runtime. In all cases, Chroma picked the same tactic plan as the ideal runtime system and the differences in the metric were due to experimental errors in the latency measurements.





The latency that was achieved by executing Face remotely and locally for all inputs on both clients is shown. In all cases, Chroma picked the option that minimized latency. This maximized the latency-fidelity metric as the fidelity was constant in all cases.

Figure 7: Relative Latency for Face

```
RPC detect_face (IN file in_image_name,
                 OUT file out_image_name);

DEFINE_TACTIC detect = detect_face;
```

Face has only one remote call (detect_face) that can be run either locally or remotely. This is captured by its single tactic.

Figure 8: Tactics for Face

5.3 Face

5.3.1 Description

Face is a program that detects human faces in images. It is representative of image processing applications of value to mobile users. Face can potentially change its fidelity by degrading the quality of the input image. However, for the purposes of this experiment, all experiments were run with full fidelity images.

Face can be run either entirely locally or entirely remotely. In both cases, it runs the exact same remote procedure and it has no other modes of operation. It thus has only one tactic and this is shown in Figure 8. Even though Face has only one tactic, this does not mean that it cannot benefit from tactics. We show in Section 7.2 how Chroma can use this single tactic to improve the performance of Face by using extra resources in the environment. Face is written in Ada and has ≈20K lines of code while the description of its tactics requires just 2 lines.

We used as input three different image files of different size (ranging from 133 KB to 621 KB in size) as inputs for the baseline experiments. There was one remote server available and both the server and the clients were unloaded.

5.3.2 Results

Figure 7 shows the latency that can be achieved when doing the face recognition locally and remotely for both configurations. Since the fidelity was constant (full quality images) in all the experiments, maximizing the latency-fidelity metric would require Chroma to pick the option that minimized the latency. We see that in all cases, Chroma chose the option that maximized the latency-fidelity metric by picking the tactic plan that minimized the latency.

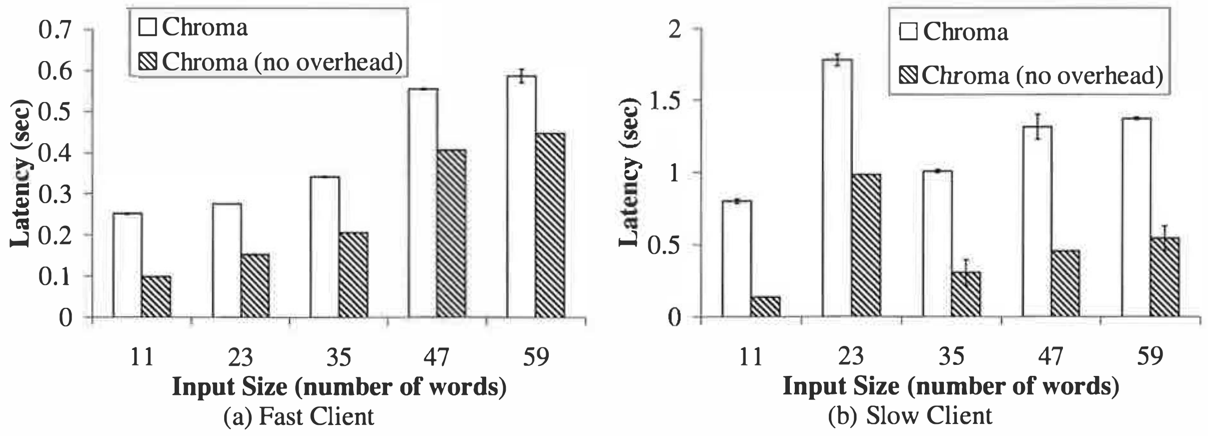
The graphs show that Face has extremely high latencies; on the order of tens of seconds per image. We will show how tactics allow us to reduce this latency without sacrificing fidelity in Section 7.2.

5.4 Summary

Sections 5.1, 5.2 and 5.3 described the performance of Chroma relative to an ideal runtime system for Pangloss-Lite, Janus and Face respectively. We see that while Chroma is not perfect, its performance is still comparable to an ideal runtime system. We believe that the results indicate that it is viable to build a tactics-based remote execution system that provides good application performance.

6 Results: Chroma's Overhead

In this section, we present the CPU overhead of Chroma's decision making using Pangloss-Lite as the example application. Pangloss-Lite has the largest number of tactic plans among all the applications used in this paper and required Chroma to do the most decision making. As such, we do not present the overhead results for the other applications as they were strictly less than the



The bars show the time needed for different tactic plans to execute with and without Chroma's decision making process. The difference in time represents the overhead of Chroma's decision making process. The tactic plans used in this experiment are the same ones Chroma chose in Figure 4 for the different inputs. The results are the average of 5 runs and are shown with 90% confidence intervals.

Figure 9: Overhead of Decision Making for Pangloss-Lite

overhead incurred for Pangloss-Lite.

Figure 9 shows the overhead of Chroma's decision making. This overhead represents the time that Chroma needs to determine the tactic plan to use. Chroma currently does not take its own overhead into account when making placement decisions and thus can achieve longer latencies than it expected. This is more apparent on slower clients as it takes longer for Chroma to make its decisions on these computationally weaker clients. From the figure, we see that Chroma's maximum overhead was less than 0.5 seconds. This overhead, while somewhat high, was still acceptable for the class of applications being targeted. We are currently improving the internal algorithms used in Chroma to reduce this overhead.

7 Results: Over Provisioning

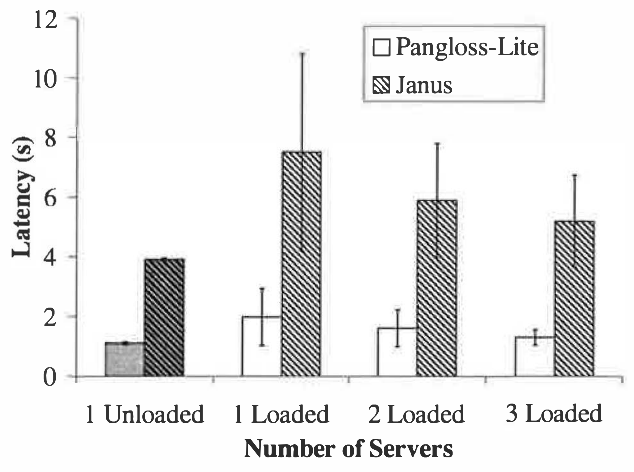
In this section, we show the performance improvements that Chroma achieves by opportunistically using extra resources in the environment. These extra resources take the form of extra available servers that can be used for remotely executing application components. We used the slow client for these experiments.

To show the benefits of this approach, we introduced an artificial load on the server that Chroma selected to remotely execute application components. This artificial load has an average load of 0.2 (i.e., on average, each CPU was utilized only 20% of the time). However, the actual load pattern itself is random. We chose a random load pattern to model the uncertainty inherent in mobile environments where remote servers could suddenly perform worse than expected due to a variety of random reasons (such as bandwidth fluctuations, extra load at the server etc.). The average load was set at 0.2 to ensure that the servers were, on average, underutilized. In contrast, a load of 0.8 (the CPU was utilized 80% of the time) or higher would indicate a heavy load.

The overall scenario we are assuming for this section is as follows; Chroma has decided where to remotely execute an application component. At the time it made the decision, Chroma noticed that the remote server was capable of satisfying the latency requirements of the operation. However, when the operation was actually executed, the actual average latency was much higher due to the random load on the server that Chroma was unaware of. We show results to quantify just how bad the average latency (and variance) becomes and how opportunistically using extra servers in the environment can help improve this. These extra servers can be used in the three ways detailed in Section 3.3 to allow us to:

- Hedge against load spikes at the remote servers: the same operation can be run on multiple servers using the "fastest result" method.
- Improve the total latency of an operation without sacrificing fidelity: the operation can be broken up into smaller parts using the "data decomposition" method and each smaller part run on a separate server.
- Satisfy absolute latency constraints of an application while providing the best possible fidelity: the operation can be run at different servers (where each server runs the operation at a different fidelity) using the "best fidelity" method and the best fidelity result that returns within the latency constraint is returned to the application.

It should be noted again that all these methods can be used automatically at runtime by Chroma without the ap-



This figure shows the use of multiple loaded servers to improve the performance of Pangloss-Lite and Janus. As we increase the numberof loaded servers, the latency and standard deviation for both applications decrease significantly and converge towards the best-case value (1 unloaded server).

Figure 10: Using Extra Loaded Servers to Improve Latency

plication being aware of them. This is one of the key benefits of using a tactics-based remote execution system.

7.1 Hedging Against Load Spikes

7.1.1 Description

This experiment shows how opportunistically using extra servers in the environments provides protection against random load spikes at any particular remote server. In this experiment, Chroma decides to execute the glossary engine of Pangloss-Lite remotely to translate a sentence containing 35 words. We ran the translation of this sentence 100 times using a different number of remote servers in parallel and noted the average latency achieved and the standard deviation.

7.1.2 Results

Figure 10 shows the results we obtained from executing the glossary engine remotely on a totally unloaded server and from executing the glossary engine remotely on one, two and three servers respectively that had the artificial load explained earlier. Figure 10 also shows the results for Janus where the recognition of utterance 5 is performed multiple times on remote servers.

The results for the totally unloaded server present the best possible average latency and standard deviation. What we notice is that when the remote server is loaded, executing the glossary engine or recognition remotely at the server results in a much higher average latency and standard deviation. We also notice that executing the glossary engine or recognition on two remote servers that are randomly loaded (with the same average load) reduces the latency and standard deviation significantly compared with the single loaded server case. Executing the glossary engine or recognition on more loaded remote servers reduces the average latency and standard deviation even further and brings them closer to the best possible results.

The reduction in latency caused by using extra servers with load was due to the load on the servers being uncorrelated. Hence, even though the average load on the servers was the same, when one server was experiencing a load spike, another server was unloaded and was able to service the request faster. Our method of using extra servers thus maximizes the probability of being able to execute the application component at an unloaded server.

This assumption of uncorrelated load is reasonable in a mobile environment for the following reasons: if the remote servers are located in different parts of the network, it is quite likely that they experience different load patterns. This is also true for remote servers that are co-located but owned by different entities. In the case where the remote servers are co-located and owed by the same entity, it is possible that they experience the same load patterns. However, in this case, enabling some sort of Ethernet-like backoff system on the remote servers will ensure that the load on each server is uncorrelated.

Of course, if every Chroma client is sending extra requests to every available server, the assumption that the load on each server is uncorrelated will not be true. We are currently studying various resource management algorithms to ensure fair usage of extra servers. We are also looking at mechanisms to allow the user to explicitly specify (if necessary) which extra servers can be used and which should not.

7.2 Reducing Latency by Decomposition

7.2.1 Description

This experiment shows how decomposing an operation into smaller pieces and executing each piece on a separate remote server reduces the overall latency of the operation. As shown in Figure 7, Face had high latencies for the three input files. However, this latency can be reduced in two ways. Firstly, the input image can be reduced in size by scaling it. However, this method reduces the fidelity of the result. The second method is to break the image into smaller pieces and separately process each piece. This method has the potential of improving the latency without reducing the fidelity.

Here, we assume that the application has previously provided Chroma with the methods for splitting and recombining the image files. Given these methods, at run-

No. of Servers Used	Average (s)	Standard Deviation (s)	Latency Reduction
1	24.54	0.05	—
2	13.59	0.05	44.6%
3	9.73	0.07	60.4%

We see that splitting the input image for the operation into smaller pieces and sending these smaller pieces to different remote servers results in a dramatic reduction in total latency. The number of servers used corresponds to the number of pieces the image file was split into.

Figure 11: Improvement in Face Latency by Decomposition

time, Chroma is able to automatically split the input images to improve application performance when extra servers become available.

7.2.2 Results

Figure 11 shows the results obtained by using this method. We ran each experiment 5 times and measured the average latency and standard deviation. The servers used were unloaded. The results show that splitting the image into smaller pieces (allowing Chroma to parallelize the operation) results in a substantial latency improvement (up to 60% reduction) over the original latency.

7.3 Meeting Latency Constraints

7.3.1 Description

Chroma allows an application to specify a latency constraint for a given operation. This is frequently required for interactive applications to meet user requirements. Chroma looks at the tactics for the application and automatically decides how to remotely execute this operation in parallel with different fidelity values for each parallel execution. For example, for Pangloss-Lite, Chroma could chose to execute the dictionary, gloss and ebmt translation engines on separate servers. When the latency constraint expires, Chroma picks the completed result with the highest fidelity and returns that to the application.

7.3.2 Results

We present results for Pangloss-Lite to show experimentally the benefits of this approach. For this experiment, we assume that the application has specified a latency constraint of 1 second. There were three remote servers available for Chroma to use. We use a sentence of 35 words as input. We load all the servers with a random load of average value 0.2. We ran each experiment 5 times.

Figure 12 shows the results for this experiment. We see that by taking the best result after 1 second and returning that to the application, Chroma is able to achieve a higher latency-fidelity metric than by waiting for all the engines to finish and returning a full fidelity result. During this experiment, Chroma did the following: It performed the translation using a different translation engine (`ebmt, gloss, dict`) on each of the three servers. When the latency constraint expired, Chroma determined which engines had successfully finished translating. Chroma then consulted the tactics description to determine how best to combine the completed results to provide the highest fidelity output. All of these steps can be done automatically by Chroma without application knowledge.

7.4 Summary

We have presented three different ways in which Chroma can use tactics to automatically improve user experience in over-provisioned environments. The improvement in each case was significant. Tactics allow us to obtain these improvements automatically at runtime without the application being aware of Chroma's decisions. The "data decomposition" method (Section 7.2), was the only method that required prior input from the application before it could be used. In this case, the application needed to tell Chroma how its data could be split into smaller pieces and recombined later. But even here, once Chroma had this information, it was able to use extra available resources to improve application performance at runtime without the application being aware of Chroma's optimizations.

8 Related Work

There have been a number of application-aware remote execution systems such as Abacus [1], Coign [3] and Condor [8]. They perform well in environments where resource availability does not change between the time the system decides how to remotely execute an application and when it actually performs the remote execution.

However, this assumption comes under fire in mobile environments. These environments are characterized by highly variable resource conditions that change on the order of seconds [5, 6, 17]. Overcoming this uncertainty requires application-specific knowledge on how to remotely partition the application.

An extra benefit of acquiring this knowledge is that it allows us to utilize additional resources in over-provisioned environments such as smart spaces with many idle compute servers. We envision that these environments will become increasingly common in the new future. Our system is designed to opportunistically

	Fidelity	Latency		Metric
		Average (s)	Standard Deviation (s)	
Running to Completion	1.0	1.96	0.15	0.51
Taking Best Result after 1s	0.75	1.00	0.01	0.77

The table shows the latencies and fidelities obtained by running all three translation engines (dict, gloss, ebmt) on the input on loaded servers. We see that taking the best result that returns before 1 second results in a higher latency-fidelity metric than using the highest fidelity result.

Figure 12: Achieving Latency Constraints for Pangloss-Lite

use these extra resources to improve application performance. We know of no other system that does this.

There have been other systems that have looked at the problem of partitioning applications. These include systems that performed object migration like Emerald [9] and systems that performed process migration [13]. Other systems [16] looked at the problem of service composition or the building of useful applications from components available in the environment. Currently, we have concentrated on the problem of identifying useful remote execution partitions of existing applications and have not performed any form of code migration or service composition.

The declarative language we use to express an application's tactics addresses some of the same issues as 4GLs [12] and 'little "languages" [4]. The latter are task-specific languages that allow developers to express higher level semantics without worrying about low level details. Our language is similar as it allows application developers to specify the remote execution capabilities of their applications at a higher level without needing to worry about low level system integration details. However, our approach is focused towards remote execution systems for mobile computing.

9 Conclusion

In this paper, we introduced the concept of tactics. This abstraction captures application-specific knowledge relevant to remote execution with minimal exposure of the implementation details. This allows the use of computationally intensive applications on handheld and wearable devices even in environments with changing resources. We showed how tactics can be used to build a remote execution system. We also provided experimental results from three applications to confirm the benefits of using tactics.

Currently, we are looking at methods of resource allocation to ensure that servers are used fairly by Chroma clients. We are also looking at various service discovery mechanisms to allow us to easily discover the presence of these servers. Finally, we are developing better software engineering methods to ease application development.

10 Acknowledgments

This research was supported by the National Science Foundation (NSF) under contracts CCR-9901696 and ANI-0081396, the Defense Advanced Projects Research Agency (DARPA) and the U.S. Navy (USN) under contract N660019928918. Rajesh Balan was additionally supported by a USENIX student research grant. We would like to thank Hewlett-Packard for donating the notebooks to be used as the servers and Compaq for donating handhelds used as clients. Finally, we would like to thank Mukesh Agrawal, Jan Harkes, Urs Hengartner, Ningning Hu, Glenn Judd, Mahim Mishra, Bradley Schmerl, Joao Sousa, the anonymous MobiSys reviewers and our shephard Marvin Theimer for their many insightful comments and suggestions related to this work. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, DARPA, USN, HP, USENIX, Compaq, or the U.S. government.

References

[1] Amiri, K., Petrou, D., Ganger, G., and Gibson, G. Dynamic function placement for data-intensive cluster computing. *Proceedings of the USENIX 2000 Annual Technical Conference*, San Diego, CA, June 2000.

[2] Balan, R. K., Sousa, J. P., and Satyanarayanan, M. Meeting the software engineering challenges of adaptive mobile applications. Technical Report CMU-CS-03-111, Carnegie Mellon University, Pittsburgh, Pennsylvania, Feb. 2003.

[3] Basney, J. and Livny, M. Improving goodput by co-scheduling CPU and network capacity. *Intl. Journal of High Performance Computing Applications*, 13(3), Fall 1999.

[4] Bentley, J. Little languages. *Communications of the ACM*, 29(8):711–21, 1986.

[5] D. Eckhardt, P. S. Measurement and analysis of the error characteristics of an in-building wireless network. *Proceeding of ACM SIGCOMM*, pages 243–254, Stanford, California, October 1996.

[6] Forman, G. and Zahorjan, J. Survey: The challenges of mobile computing. *IEEE Computer*, 27(4):38–47, April 1994.

[7] Frederking, R. and Brown, R. D. The Pangloss-Lite machine translation system. *Expanding MT Horizons: Proceedings of the Second Conference of the Association for*

Machine Translation in the Americas, pages 268–272, Montreal, Canada, 1996.

[8] Hunt, G. C. and Scott, M. L. The Coign automatic distributed partitioning system. *Proceedings of the 3rd Symposium on Operating System Design and Implemetation (OSDI)*, pages 187–200, New Orleans, LA, Feb. 1999.

[9] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the emerald system. *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, volume 21, pages 105–106, 1987.

[10] Katz, R. H. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1(1):611–17, 1994.

[11] Lee, C., Lehoczky, J., Siewiorek, D., Rajkumar, R., and Hansen, J. A scalable solution to the multi-resource QoS problem. *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS '99)*, pages 315–326, Phoenix, AZ, Dec. 1999.

[12] Martin, J. *Fourth-Generation Languages*, volume 1: Principles. Prentice-Hall, 1985.

[13] Milojicic, D. S., Douglis, F., Paindaveine, Y., Wheeler, R., and Zhou, S. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[14] Narayanan, D. and Satyanarayan, M. Predictive resource management for wearable computing. *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys)*, San Francisco, CA, May 2003.

[15] Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R. Agile application-aware adaptation for mobility. *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 276–287, Saint-Malo, France, October 1997.

[16] Raman, B. and Katz, R. An architecture for highly available wide-area service composition. *Computer Communications Journal, special issue on 'Recent Advances in Communication Networking'*, May 2003.

[17] Satyanarayanan, M. Fundamental challenges in mobile computing. *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, Philadelphia, PA, May 1996.

[18] Satyanarayanan, M. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, Aug. 2001.

[19] Satyanarayanan, M. The evolution of coda. *ACM Transactions on Computer Systems (TOCS)*, 20(2):85–124, 2002.

[20] Schneiderman, H. and Kanade, T. A statistical approach to 3d object detection applied to faces and cars. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 746–751, Hilton Head Island, South Carolina, June 2000.

[21] Sun Microsystems Inc. *Remote Method Invocation Specification.*

[22] Waibel, A. Interactive translation of conversational speech. *IEEE Computer*, 29(7):41–48, July 1996.

[23] Weiser, M. The computer for the twenty-first century. *Scientific American*, pages 94–101, September 1991.

Collaboration and Multimedia Authoring on Mobile Devices

Eyal de Lara◇, Rajnish Kumar†, Dan S. Wallach†, and Willy Zwaenepoel‡

◇Deparment of Computer Science
University of Toronto
delara@cs.toronto.edu

†Department of Computer Science
Rice University
{rajnish,dwallach}@cs.rice.edu

‡School of Computer and Communication Science
Ecole Polytechnique Federale de Lausanne, Switzerland
willy.zwaenepoel@epfl.ch

Abstract

This paper introduces *adaptation-aware editing* and *progressive update propagation*, two novel mechanisms that enable authoring multimedia content and collaborative work on mobile devices. Adaptation-aware editing enables editing content that was adapted to reduce download time to the mobile device. Progressive update propagation reduces the time for propagating content generated at the mobile device by transmitting either a fraction of the modifications or transcoded versions thereof.

With application-aware editing and progressive update propagation, an object present at a mobile device is characterized not only by a particular version, as in conventional replication, but also by a particular fidelity. We demonstrate that replication models can be extended to account for fidelity independently of the mechanisms used for concurrency control and consistency maintenance. As a result, the two techniques described in this paper can easily be added to any replication protocol, whether optimistic or pessimistic.

We report on our experience implementing adaptation-aware editing and progressive update propagation. Experiments with two multimedia applications, an email reader and a presentation software package, show that both mechanisms can be added with modest programming effort and achieve substantial reductions in upload and download latencies.

1 Introduction

Research on mobile computing has made significant progress in adapting applications for viewing multimedia content on mobile devices [5, 8, 21]. Multimedia authoring and collaborative work on these platforms remain, however, open problems.

We identify three factors that hinder multimedia authoring and collaborative work over bandwidth-limited links:

1. Read adaptations. The adaptation techniques used to lower resource usage (e.g., energy, bandwidth) may result in situations where content present at the mobile device differs significantly from the versions stored at the server. Typical adaptation techniques adapt by downloading just a fraction of a multimedia document, or by trancoding content into lower-fidelity representations. Naively storing user modifications made to an adapted document may delete elements that were not present at the mobile device, or it may replace high-fidelity data with the transcoded versions sent to the mobile device (even in cases where the user did not modify the transcoded elements).
2. Large updates. Mobile users can generate large multimedia content (e.g., photographs, drawings, audio notes) whose propagation may result in large resource expenditures or long upload latencies over a bandwidth-limited link .
3. Conflicts. The use of optimistic replication models [12, 23] allows concurrent modifications that may conflict with each other. Conflicts can occur in other circumstances as well, but low bandwidth and the possibility of frequent disconnection make their occurrence more likely.

This paper introduces *adaptation-aware editing* and *progressive update propagation*, two novel mechanisms that enable document authoring and collaborative work over bandwidth-limited links. These mechanisms extend traditional replication models to account for the fidelity level of replicated content. Both mechanisms decompose multimedia documents into their component structure (e.g., pages, images, sounds, video), and keep track of consistency and fidelity at a component granularity. Adaptation-aware editing enables editing adapted documents by differentiating between modifications made by the user and those that result from adaptation. Progressive update propagation reduces the time and the resources required to propagate components created or modified at the bandwidth-limited device by transmitting subsets of the modified components or transcoded versions of those modifications. Adaptation-aware editing and progressive update propagation also reduce the likelihood of update conflicts in two ways. First, by working at the component level rather than the whole-document level, they reduce the sharing granularity. Second, because both mechanisms lower the cost to download and upload component data, they encourage more frequent communication, hence increasing the awareness that users have of their collaborators' activities [3].

By reducing the cost of propagating multimedia content, adaptation-aware editing and progressive update propagation enable new types of applications and extend the reach of existing applications into the mobile realm. The following two examples illustrate the use of both mechanisms:

1. *Maintenance.* A work crew inspects damage to a plant caused by an explosion. They use a digital camera to take pictures of the problem area, and send the pictures over a wireless connection to the head office. Since bandwidth is low, and they want an urgent assessment of the seriousness of the situation, they use progressive update propagation to initially send low-resolution versions of the pictures. These initial images allow the head office to determine quickly that there is no need to declare an emergency, but that repair work nonetheless needs to be started immediately. The crew continues to use progressive update propagation to send higher-resolution versions of the pictures, sufficiently detailed to initiate repairs. The head office forwards these pictures to a trusted contractor and to the insurance company. The contractor uses adaptation-aware editing to indicate the suggested repairs on the pictures, and sends the marked-up pictures back to the head office and the insurance company. Both approve the repairs, and the contractor heads out to the site. When the work crew arrives back at the office, full-resolution pictures are saved for later investigation.
2. *Collaborative presentation design.* A team member on a mobile device takes advantage of adaptation-aware editing to reduce download time by downloading and editing an adapted version of a presentation. The adapted document consists of just a few slides of the original presentation and has low-fidelity images, sounds, and videos. The team member then uses progressive update propagation to share her modifications to the presentation, which include a photograph taken with a digital camera. Progressive update propagation reduces the time for uploading the photograph by sending a low-fidelity version of the image. When the team member reconnects over a high-bandwidth link, the system automatically upgrades the version of the photograph.

The previous scenarios cannot be handled by current adaptation systems that only handle adaptation of read-only content. They also cannot be supported by current replication systems. Propagating transcoded versions of components as described in the above examples, requires the replication model to account for the fidelity level of replicated content. Upgrading the fidelity of an image in a particular version of a document is different from creating a new version with (user) modifications to the document.

This paper shows that fidelity can be added to a replication protocol independently of the mechanisms used for concurrency control and consistency maintenance. Replication models are typically represented by state diagrams, and we follow this general paradigm. We present state diagrams that incorporate the presence of transcoded versions of components, for use with both optimistic and pessimistic replication. The introduction of transcoded component versions is orthogonal to the maintenance of consistency between replicas. More specifically, new states are added to represent transcoded versions, but the semantics of the existing states and the transitions between them remain unchanged. Therefore, fidelity can be added easily to any replication protocol, whether optimistic or pessimistic.

There are several possible implementations of adaptation-aware editing and progressive update propagation. We present a prototype implementation of these mechanisms that takes advantage of existing run-time APIs and structured document formats [5]. This implementation allows us to adapt applications for multimedia authoring and collaboration *without* changing their source code.

We demonstrate our implementation by experimenting with the Outlook email browser and the PowerPoint presentation software. Both applications see large reductions in user-perceived latencies. For Outlook, progressive update propagation reduces the time a wireless author has to stay connected to propagate emails with multimedia attachments. For PowerPoint, adaptation-aware editing and progressive update propagation reduce the time that wireless collaborators need to wait to view changes made to the presentation by their colleagues.

The rest of this paper is organized as follows. Section 2 introduces adaptation-aware editing and progressive update propagation and explores the implications of extending pessimistic and optimistic replication models to support these mechanisms. Sections 3 and 4 present the design and evaluation of our prototype implementation of adaptation-aware editing and progressive update propagation. Finally, Sections 5 and 6 discuss related work and conclude the paper.

2 Incorporating Fidelity in Replication Protocols

In this section, we describe the implications of extending traditional replication models to provide various degrees of support for adaptation-aware editing and progressive update propagation.

We assume that it is possible to decompose documents into their component structure (e.g., pages, images, sounds, video). Component decomposition can be guided by the document's file format, or by a policy set by the content provider or by the client. For example, HTML documents [24], as well as documents from popular productivity tools [4] use well defined tags to signal the presence of multimedia elements such as images, sounds, and videos.

Multiple versions of a component can co-exist in different replicas. Two versions of a component may differ because they have different creation times, and hence reflect different stages in the development of the component, or because they have different fidelity levels. We consider two fidelity classes: *full* and *partial*. For a given creation time, a component can have only one full-fidelity version but many partial-fidelity versions. A component is present at full fidelity when its version contains data that is equal to the data when the version was created. Conversely, a component is present with partial fidelity if it has been lossily transcoded from the component's original version. Fidelity is by nature a type-specific notion, and hence there can be a type-specific number of different partial-fidelity versions. We assume that it is possible to determine whether one version has higher fidelity than another one.

This discussion considers both pessimistic and optimistic replication models. A pessimistic replication model guarantees that at most one replica modifies a component at any given time, and that a replica does not modify a component while it is being read by some other replica. The mutual exclusion guarantee can be realized by various mechanisms, such as locks or invalidation messages. With optimistic replication, replicas may read and write components without any synchronization. A manual or automatic reconciliation procedure resolves conflicts caused by concurrent writes on different replicas.

Replication models are typically represented by state diagrams, and we follow this general paradigm. The states and transitions for a replication model are independent of the specific mechanisms used for consistency maintenance and depend only on whether the replication model is pessimistic or optimistic. The discussion in this section is therefore independent of specific mechanisms used for consistency maintenance, such as invalidations, leases, or timeouts. The mechanisms for consistency maintenance only determine what events trigger specific transitions (e.g., transition from Clean to Empty on receiving an invalidation message). This discussion is also independent of the specific mechanisms used to propagate versions between replicas. The use of data or operation shipping, as well as optimizations, such a version diffing, are implementation decisions that do not affect the underlying replication model.

We consider both primary replica and serverless approaches. In a primary replica approach, a server holds the primary replica of the document. Clients can replicate subsets or all of the document's components by reading them from the server's primary replica. Client modifications are sent to the server, and there is no direct communication between clients. In contrast, in a serverless configuration there is no centralized server or primary replica and replicas communicate directly.

In the rest of this section, we first describe the implications of supporting adaptation-aware editing and progressive update propagation in isolation. We then describe replication models that support both mechanisms. The initial discussion assumes a primary replica. Serverless systems are discussed afterwords.

2.1 Adaptation-Aware Editing

The simplest form of adaptation-aware editing limits users to modifying only components that are loaded with full fidelity at the bandwidth-limited device. Such an implementation requires the replication system to keep track of which components are available at the bandwidth-limited device and whether these components have been transcoded into partial-fidelity versions. This information is normally already present in replication systems or can be easily added. The replication system then prevents users from modifying any component that is not present with full fidelity.

A simple extension to the previous model is to allow users to (completely) overwrite or delete partial-fidelity components or components that were not included in the client's replica subset. In this scenario, the user can (completely) replace the content of a component that was not loaded or that was loaded at partial fidelity with new full-fidelity content generated at the bandwidth-limited device. The user can also remove a component from the document altogether. Adding this functionality does not require keeping extra state.

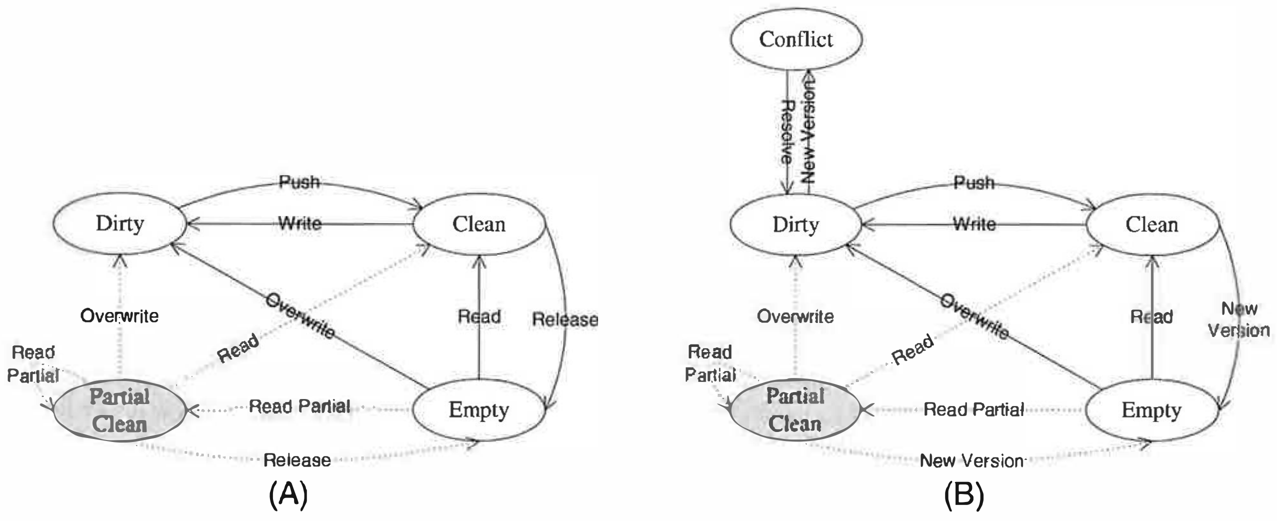


Figure 1: State transition diagram for a pessimistic (A) and an optimistic (B) replication model with support for adaptation-aware editing. Partial-fidelity states and the transitions in and out of these states are represented with gray ovals and dotted arrows, respectively. In contrast, states present in traditional replication models and their transitions are represented with clear ovals and full arrows, respectively.

2.1.1 Pessimistic Replication

Figure 1(A) shows the state transition diagram for individual components of a client replica for a pessimistic replication model that supports modifying full-fidelity component versions and overwriting partial-fidelity component versions. In our state diagrams, we represent new partial-fidelity states by gray ovals and the new transitions in and out of these states by dotted arrows. In contrast, we represent the states present in traditional replication models by clear ovals and their transitions by full arrows. The state diagram for the primary replica (not shown) stays the same as without support for adaptation-aware editing. This diagram contains two states, Empty and Clean, with the obvious meanings.

In the client replica state transition diagram, a component can be in one of four states: Empty, Partial-Clean, Clean, and Dirty. A component is in Empty when it is being edited by some other client replica or when the client chooses not to read it. A component transitions into Partial-Clean when the client replica reads a partial-fidelity version. This version can be further refined by reading higher-fidelity partial-fidelity versions (i.e., Read-Partial) or the component can transition into Clean by reading a full-fidelity version. The component transitions into Dirty when the client replica either modifies a full-fidelity version (i.e., component in Clean state) or overwrites an unloaded component or a partial-fidelity version (i.e., component in Empty or Partial-Clean). The component transitions back to Clean when the client replica propagates a full-fidelity version to the primary replica. Finally, a component transitions back to Empty when the client replica no longer wishes to read the component. Transitions to Empty depend on the specific mechanisms used to guarantee mutual exclusion, and can occur, for example, when the client replica releases a lock or receives an invalidation.

2.1.2 Optimistic Replication

Figure 1(B) shows the state transition diagram for an optimistic replication model. The optimistic replication diagram differs from the pessimistic diagram (Figure 1(A)) in two ways: First, it has an extra state for conflict resolution. The component transitions to the Conflict state when the replica detects the primary replica version and the client replica version are concurrent (i.e., it is not possible to determine a partial ordering for the two versions) [15]. The component transitions back to the Dirty state once the client replica reads the conflicting version and resolves the conflict. Second, transitions from the Clean and Partial-Clean states to the Empty state occur when the client replica learns that the primary replica has a more recent version for the component. The decision of when to transition to Empty is left to the implementation. Some implementations may eagerly invalidate the current version, while others may allow the user to keep working with the current version. In other words, it is the implementation's responsibility to decide how eagerly it wants to act on the consistency information it receives.

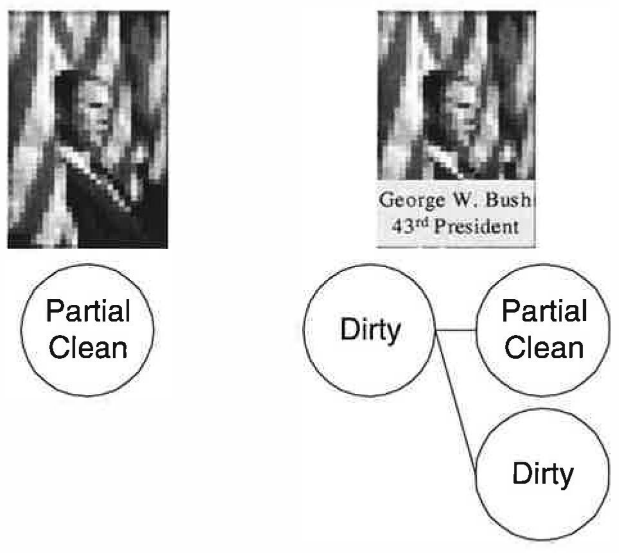


Figure 2: A component is split after modifications to a partial-fidelity version.

2.1.3 Modification of Partially-Loaded Components

A more ambitious form of adaptation-aware editing allows users to modify just a portion of a partial-fidelity version, for example, to replace parts of transcoded images, audio recordings, or video streams. Such modifications result in a component version that contains a mixture of partial- and full-fidelity data, which contravenes our initial assumption that the replication system keeps track of fidelity at the component granularity. While the semantics of some data types, such as images, may support modifications to just parts of the component's version, these semantics are not visible to the replication system. To reflect the changes to the replication system, the component has to be split into two subcomponents as shown on Figure 2. The first subcomponent holds the partial-fidelity data, which was not modified by the user, and the second subcomponent holds the full-fidelity modifications made by the user. The original component (now turned into a container for the two subcomponents) transitions to the Dirty state, to reflect the change in the document's component structure. The subcomponent holding unmodified partial fidelity data remains in the Partial-Clean state. In contrast, the subcomponent holding new full-fidelity data transitions to the Dirty state.

A partial-fidelity version can also be changed by an operation that does not produce any full-fidelity data, for instance, by applying a gray-scale filter to a partial-fidelity image. For these cases, the operation rather than the resulting data has to be propagated to the primary replica, and applied there [16]. After the operation has been propagated and applied to the server's version, the client's version transitions into Partial-Clean if the client's version can be lossily trancoded from the server's version, and to Empty otherwise. In other words, the client's version transitions to Partial-Clean only if the lossy trancoding algorithm used to derive the client's version and the operation being propagated are commutative.

In either of the above cases, reflecting the modifications made to the partial-fidelity version on the full-fidelity version available at the server requires data-type specific instrumentation (i.e., code that knows how to extract the modifications to a partial-fidelity version, and merge them with the full-fidelity version at the server).

2.2 Progressive Update Propagation

A replication system supports progressive update propagation by propagating a subset of the modified components and/or by propagating partial-fidelity versions of modified components.

In this section, we consider the implications of an implementation that supports progressive update propagation but does not support transcoding components on read or editing partial-fidelity components. In such an implementation, client replicas have by default full-fidelity versions of the components they replicate. A client replica has a partial-fidelity version for a component only when the component is being updated by some other client and the updates are being progressively propagated. In other words, the decision to propagate partial-fidelity data is made by the replica that is writing the component and not by the reader, as was the case in the previous section. Moreover, independently of whether we implement a pessimistic or optimistic approach to replication, once a partial-fidelity version has been propagated to the primary replica, it can only be replaced with another version created by the same writer (i.e., a higher-fidelity version or a more recent version). Replacing a partial-fidelity version with a version created by a different writer would require editing partial-fidelity components, which is not allowed by the implementation discussed in this section. In Section 2.3 we describe an implementation that allows a writer to replace a partial-fidelity version created by another writer.

2.2.1 Pessimistic Replication

Supporting progressive update propagation requires adding one new state to the primary replica's state transition diagram (Partial-Clean) and two new states to the client replica's state transition diagram (Pseudo-Dirty and Partial-Clean).

Figures 3 (A) and (B) show the state transition diagram for an individual component in a pessimistic replication model at a client replica and at the primary replica, respectively. The transition diagram for the primary replica

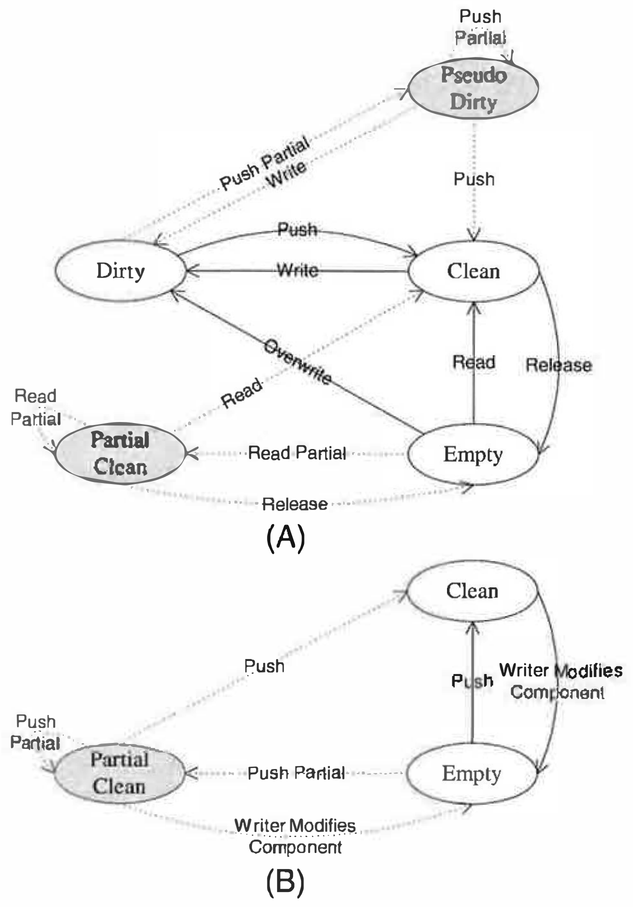


Figure 3: Pessimistic replication state transition diagram for components of the client (A) and primary (B) replicas.

is simple. A component at the primary replica can be in one of three states: Empty, Partial-Clean, and Clean. A component is in the Empty state while it is being edited by a client replica. The component transitions to the Partial-Clean and Clean states when the writer pushes a partial-fidelity or a full-fidelity version of the components, respectively.

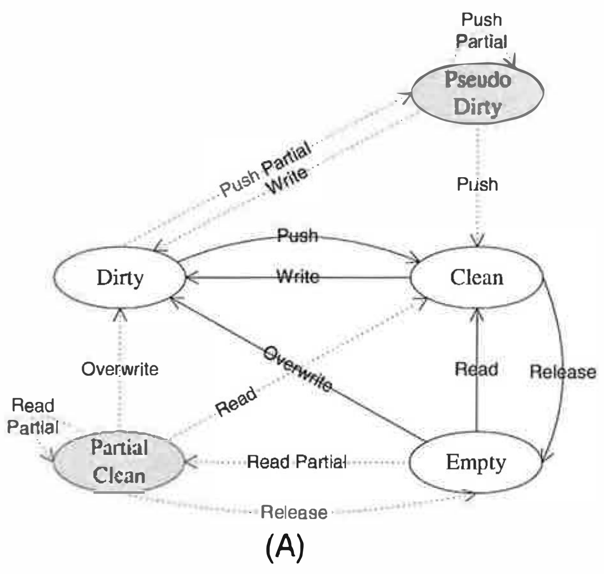
A component in a client replica can be in one of five states: Empty, Clean, Partial-Clean, Dirty, and Pseudo-Dirty. A component is in the Empty state either while it is being modified by some other client replica or when the client has chosen not to read it. A component transitions to Clean by reading a full-fidelity version. If only a partial-fidelity version is available at the primary replica because the last writer has not propagated a full-fidelity version yet, the client replica can read this version and transition to Partial-Clean. The component transitions from Clean to Dirty after the client modifies its content. The client can then propagate modifications to the primary replica in two ways. First, the writer can push a full-fidelity version of the modifications, forcing the component at the primary and writer's replica to transition to Clean. Second, the writer can propagate a partial-fidelity version of the component, forcing the component to transition to Partial-Clean in the primary replica, and to Pseudo-Dirty in the writer's replica. The various replicas remain in these states until the writer pushes a full-fidelity version and the component at both the primary and writer's replica transition to Clean. At this time, other client replicas can read the full-fidelity version and transition to Clean. Alternatively, a writer in Pseudo-Dirty can modify the component for a second time and transition to Dirty. If other replicas are to obtain access to a partial-fidelity version, it is imperative that the writer relinquishes exclusive access rights. This enables other replicas to read the partial-fidelity version, but requires the replica in Pseudo-Dirty to re-acquire exclusive access to the component before it can modify it again and transition to Dirty.

2.2.2 Optimistic Replication

In an optimistic replication scheme, before propagating modifications to a component, the writer has to determine if his modifications conflict with other modifications previously reflected at the primary replica. If there is a conflict, the client replica has to resolve it by merging (in a type-specific way) the full-fidelity versions of the conflicting modifications. After resolution, the client replica can propagate a full- or partial-fidelity version of the component to the primary replica. If, however, the primary replica has only a partial-fidelity version for a conflicting component (i.e., the concurrent writer has not propagated a full-fidelity version of its modifications), the two versions cannot be merged as this would violate the restriction on editing partially-loaded components. In this case, conflict resolution has to be delayed until the client replica, which propagated the conflicting partial-fidelity version, propagates a full-fidelity version of its modifications. This problem demonstrates the limitations of implementing partial update propagation for optimistic concurrency control in the absence of adaptation-aware editing. The next section describes how to implement this combination.

2.3 Combining Adaptation-Aware Editing and Progressive Update Propagation

In this section we explore the implications of extending pessimistic and optimistic replication models to support both partial document editing and progressive update propagation. We consider replication systems that support all the features presented in Sections 2.1 and 2.2.



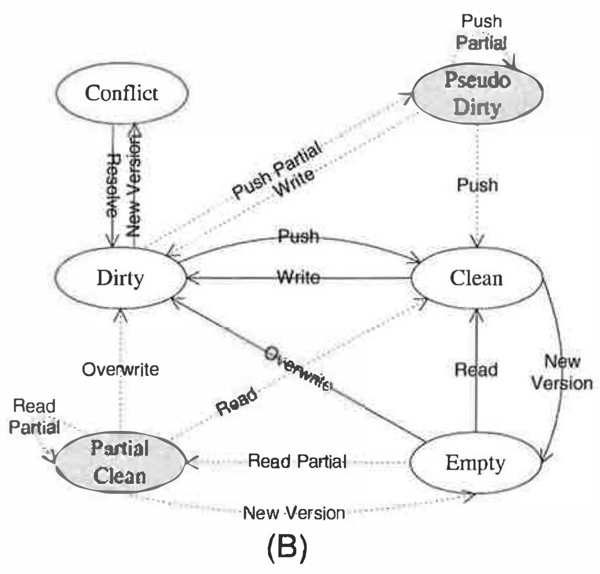


Figure 4: State transition diagrams for individual components of client replicas that support partial document editing and progressive update propagation based on pessimistic (A) and optimistic (B) replication models.

2.3.1 Pessimistic Replication

Figure 4 (A) shows the state transition diagram for components at the client replica for a pessimistic replication system that supports adaptation-aware editing and progressive update propagation. The state transition diagram for components at the primary replica is the same as the one shown in Figure 3 (B).

The diagram in Figure 4 (A) is similar to that of Figure 3 (A) and most states have similar semantics. The semantics of Partial-Clean and Partial-Dirty are, however, a little different. A component may be in the Partial-Clean state because the client requested a partial-fidelity version to reduce its network usage, or because only a partial-fidelity version of the component is available at the primary replica. As was the case in Section 2.2.1, if other replicas are to obtain access to a partial-fidelity version, it is imperative that the writer relinquishes exclusive access rights. Moreover, if another replica is to replace the partial-fidelity version with a later version with new data, the current writer should relinquish all access.

Because adaptation-aware editing is supported, a second client replica can read and modify a component as soon as a partial-fidelity version is available at the primary replica. Two scenarios are possible. First, the second client replica can delete or completely overwrite the component. In this case, the second writer propagates the new version of the component to the primary replica (in either full or partial fidelity), where it supersedes all previous versions, including any version propagated by the first writer. Based on the implementation, any further versions propagated by the first writer are either stored for archival purposes or discarded. Second, the second client replica modifies just a portion of the component. As was the case in Section 2.1.3, propagating the modifications to the partial-fidelity version requires data-type specific instrumentation. This instrumentation may or may not require waiting for the first writer to propagate a full-fidelity version of its modifications. Alternatively, for some data types it may be possible to propagate the second writer's modifications to the primary replica, and merge them lazily with the first writer's modification as they arrive.

2.3.2 Optimistic Replication

Figure 4 (B) shows the state transition diagram for an optimistic replication model that supports partial document editing and progressive update propagation. The state transition diagram is similar to that of the pessimistic replication model we discussed in Section 2.3.1, with states and transitions with the same names having equivalent semantics. The optimistic replication diagram differs in two ways: First, it has an extra state for conflict resolution. Second, it transitions from Clean and Partial-Clean to Empty and from Dirty to Conflict when the client replica learns about a more recent or concurrent component version. As was the case in Section 2.1.2, the eagerness with which the transitions to the Empty state are taken is an implementation decision.

Supporting both adaptation-aware editing and progressive update propagation, also enables client replicas to resolve conflicts even when the server's primary-replica just has a partial-fidelity version for the component. In such case, the client replica reads the conflicting partial-fidelity version, resolves the conflict, and chooses whether to propagate a full- or partial-fidelity version of the modifications. Resolving conflicts using partial-fidelity versions requires data-type specific functionality similar to that described in Section 2.1.3 for reflecting modifications made to partial-fidelity versions.

2.4 Serverless Replication

The earlier state diagrams can be carried over from a primary replica configuration to a serverless configuration without any change. In a serverless configuration, when a replica modifies a component it becomes the source for distributing these modifications to other replicas. In practice, however, not all replicas have to read the modifications directly from the source replica and replicas can get these modifications from some other replica that in turn got the modifications from the source replica.

Independently of how the modifications are propagated, the last writer has a full-fidelity version of the component and is perceived by other replicas as the source for this version. Hence, the replica that writes the component last becomes effectively a temporary "primary replica" for the component that it modified. The states and state transitions otherwise remain the same. If a replica wants to progressively read the component from the primary replica, it needs to maintain a Partial-Clean state. If the temporary primary replica for a particular component wishes to progressively propagate its modifications (i.e., in a push-based implementation), it needs to maintain a Pseudo-Dirty state. If it wants to update more than one replica concurrently, it needs to maintain the progress of each individual transmission as part of that state.

2.5 Summary

We have described the changes necessary to the state diagrams for pessimistic and optimistic replication models in order to support adaptation-aware editing and progressive update propagation. In general, the changes involve adding states and transitions. The existing states and transitions remain with their original semantics. Some complications arise if we allow modifications to partial-fidelity versions, requiring components to be split to reflect old partial-fidelity data and new full-fidelity data. Additionally, data-type specific instrumentation may be required to extract the modifications and reflect them on the full-fidelity version.

3 The CoFi Prototype

This section describes CoFi, a prototype implementation of adaptation-aware editing and progressive update propagation. We named our prototype CoFi because it keeps track of both *co*nsistency and *fi*delity. We first discuss CoFi's system architecture. We then present our optimistic primary replica implementation of adaptation-aware editing and progressive update propagation (see Section 2.3.2).

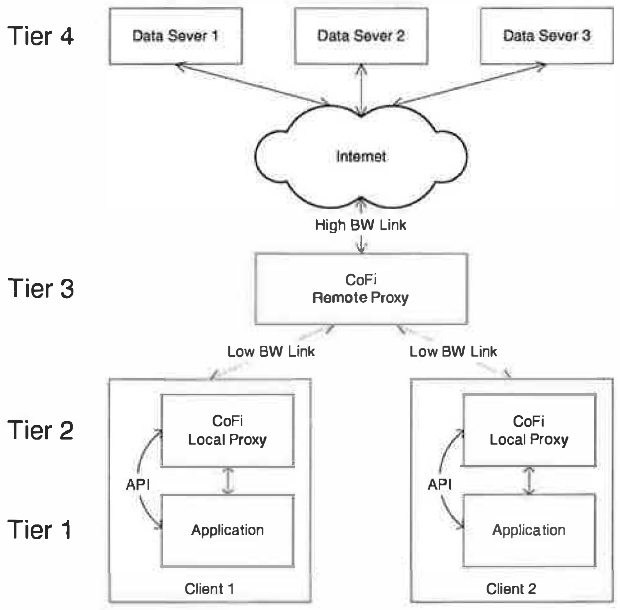


Figure 5: CoFi architecture.

3.1 System Architecture

CoFi adapts applications for collaborative and multimedia authoring over bandwidth-limited networks *without* modifying their source code or the data repositories. CoFi follows the philosophy introduced in Puppeteer for read-only adaptation [5], which takes advantage of the exposed run-time APIs and structured document formats of modern applications.

Figure 5 shows the four-tier CoFi system architecture. It consists of the application(s), a local and a remote proxy, and the data server(s). The application(s) and data server(s) are completely unaware of CoFi. Data servers can be arbitrary repositories of data such as Web servers, file servers, or databases. All communication between the application(s) and the data server(s) goes through the CoFi local and remote proxies that work together to implement adaptation-aware editing and progressive update propagation. The CoFi local proxy runs on the bandwidth-limited device and manipulates the running application through a subset of the application's exported API. The local proxy is also in charge of acquiring user modifications, transcoding component versions, and running the adaptation policies that control the download and upload of component versions. The CoFi remote proxy runs on the

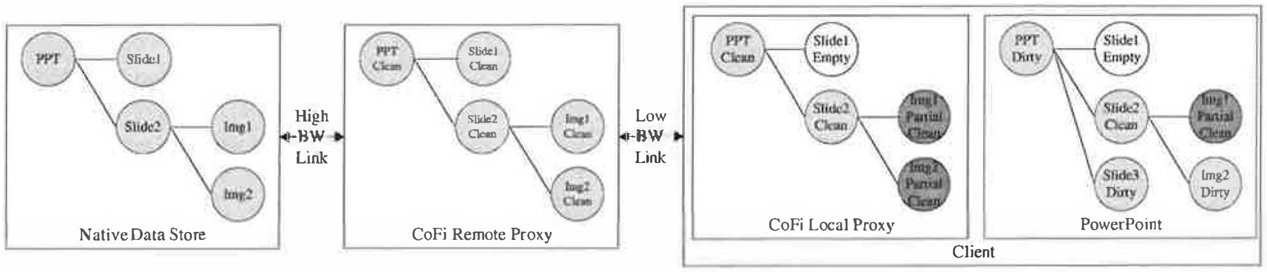


Figure 6: The native data store, CoFi remote and local proxies, and the application can have versions of the document that differ in their component subsets and fidelities.

other side of the bandwidth-limited link and is assumed to have high-bandwidth and low-latency connectivity (relative to the bandwidth-limited device) to the data servers. The CoFi remote proxy is responsible for interacting with the native store and transcoding component versions. Because applications differ in their file formats and run-time APIs, the CoFi proxies rely on component-specific drivers to parse documents and uncover their component structure, to detect user modifications, and to interact with the application's run-time API.

CoFi supports subsetting and versioning adaptation policies. Subsetting policies communicate a subset of the elements of a document, for example, the first page. Versioning policies transmit a less resource-intensive version of some of the components of a document, for example, a low-fidelity version of an image. CoFi adapts applications by extracting subsets and versions from documents. CoFi uses the exported APIs of the applications to incrementally increase the subset of the document or improve the fidelity of the version of the components available to the application. For example, it uses the exported APIs to insert additional pages or higher-fidelity images into the application.

3.2 Optimistic Primary Replica Replication

CoFi implements an optimistic primary replica replication model as described in Section 2.3.2. The prototype consists of a group of bandwidth-limited nodes, each running a CoFi local proxy, that collaborate by exchanging component data over a single CoFi remote proxy, which stores the primary replica of the document. When a document is first opened, it is imported from its native data store into the CoFi remote proxy. Further accesses to the document, both reads and writes, are then served from the CoFi remote proxy's version. In a more complete implementation, there would be multiple remote proxies communicating between each other, but this communication can be implemented by known methods and the current prototype allows us to focus on the novel aspects of CoFi. To enable communication with non CoFi-enabled programs, CoFi exports document modifications back to their native data store.

In CoFi, several versions of a document co-exist in various parts of the system. Figure 6 exemplifies the state of the system for a single client editing a PowerPoint document. The figure shows that there is one version of the document in each of the native data store, the CoFi remote and local proxy, and the application. Moreover, the figure shows that these versions differ in their component subsets and component fidelities. In the example, the native data store and the CoFi remote proxy have complete versions of the document. In contrast, both the CoFi local proxy and the application have just incomplete versions: the first slide is empty and the images of the second slide are only present in partial fidelity. Finally, the application version has an extra slide component.

The differences between the versions in the remote and local proxy result from subsetting and versioning adaptations. In contrast, the differences between the versions in the local proxy and the application result from user modifications. The native store and remote proxy versions can differ because the two versions have not been synchronized (i.e., modifications have not been propagated to the native store), or when the native store version is being updated using out-of-band mechanisms (i.e., outside of the CoFi system).

The rest of this section describes how versions of the document converge by exchanging component data. First, we describe how CoFi propagates user modifications from the bandwidth-limited device to the CoFi remote proxy and the data store. Second, we describe how the CoFi local proxy refreshes the application's document version with newer or higher-fidelity component versions.

3.2.1 User Modification Propagation

Adaptation policies running on the CoFi local proxy (as described in Section 3.1) control the propagation of user modifications to the CoFi remote proxy. Update propagation involves four stages: acquiring user modifications, resolving conflicts, transmitting modifications to the remote proxy, and synchronizing the modifications with the document's native store.

Acquire Modifications CoFi acquires user modifications by comparing the local proxy's document version to the application's version. Ideally, CoFi would use the application's exported API to acquire any user modifications. When such functionality is not provided by the application's API, CoFi instructs the application to save a temporal version of the document in the local file system. CoFi then parses the temporary document and compares it to the local proxy version.

Conflict Resolution CoFi detects conflicting modifications by tagging component versions with version numbers, which determine the partial order of modifications in the system. CoFi implements both client- and server-based conflict resolution. In client-based resolution, the local proxy fetches the conflicting version from the remote proxy, resolves the conflict and creates a new version that dominates the two conflicting versions. In server-based resolution, the client pushes its version to the remote proxy, and a resolver executing in the remote proxy creates a new version that merges the conflicting modifications.

When user intervention is necessary to resolve a conflict, conflict resolution is client-based. To facilitate conflict resolution, the application-specific resolution policy can use the application's exported API to present the conflicting component versions in the context of the application's environment.

Modification Transmission A policy running on the local proxy selects the subset of components for which to propagate modifications, as well as the fidelity level for each component in the subset. The policy can later increase the fidelity level of a previously propagated component by re-selecting the component and pushing a higher-fidelity version.

Synchronization with Native Storage CoFi's remote proxy exports documents to their native storage to enable information sharing with clients outside of the CoFi system and to leverage the mechanisms that these storage systems may implement (e.g., availability, fault tolerance, security, etc). Before exporting modifications, CoFi needs to detect if the native store has been modified by an application outside of CoFi's control. CoFi detects and resolves conflicts created by out-of-band modifications using similar mechanisms to those explained above (i.e., by comparing the last-known and current states of the native store).

3.2.2 Refreshing the Application's Version

Adaptation policies can refresh the application's document version to reflect changes made by other users or to increase the fidelity of a component present at partial fidelity. This process involves three steps: fetching newer or higher-fidelity versions, detecting any user modifications to the components about to be updated, and using the application's API to update the application's document version. If the update process detects that the components have been modified by the user, then a conflict has occurred and the modifications have to be merged with the new version fetched from the remote proxy in a component-specific way – following the techniques for conflict resolution described in Section 3.2.1.

3.3 Implementation Details

3.3.1 User Interaction with CoFi

In our current prototype, the applications' toolbars are extended with extra fields for selecting an adaptation policy that determines the fidelity level at which a document is opened or saved. Eventually, CoFi could rely on monitoring of bandwidth or other resources to automatically choose a particular fidelity level.

CoFi also provides a *Component Viewer* window that shows the current state of components in a document. Using this window, users can determine what components are currently loaded in the application, what components are in progress of being loaded, whether modifications to a component have been propagated to the remote proxy and with what fidelity, whether a newer version of a component is known to be available at the remote proxy, and whether a conflict has been detected for any component. Users can also interact with the Component Viewer to control the propagation of component versions.

3.3.2 Client-Server Interactions

The current prototype implementation is client-driven. That is, clients specify when they want to read or write a document and at what fidelity. The client also indicates when it wants to get or send a refinement of an earlier transcoded version. The server does not notify the clients of new versions or new refinements. Such a facility could be added through a callback mechanism, but would leave other aspects of the implementation unchanged.

3.3.3 Relationship to Puppeteer

CoFi shares some of the code base of Puppeteer [5], namely the code to parse document formats, some of the code in the local proxy to interact with the application, and the protocol to interact between the local and remote proxies.

4 Experimental Results

In this section, we report on our experience using CoFi to add adaptation-aware editing and progressive update propagation to the Outlook email client and the PowerPoint presentation system. We implement progressive update propagation for both applications. For PowerPoint we also support adaptation-aware editing. The CoFi drivers and policies we implement for Outlook and PowerPoint consist of 2,365 and 3,315 lines of Java code, respectively.

We measure the performance of CoFi on an experimental platform consisting of three 500 MHz Pentium III machines running Windows 2000. Two of the machines are configured as clients and one as a server. Client machines run the user application and the CoFi local proxy. The server machine runs the CoFi remote proxy. Clients and the CoFi server communicate via a fourth PC running the DummyNet network simulator [26]. This setup allows us to control the bandwidth between clients and server to emulate various network technologies. We use our departmental NFS and IMAP servers as the native stores for our experiments with PowerPoint and Outlook, respectively.

4.1 Outlook

We developed an email service that supports the progressive propagation of images embedded in or attached to emails. On the sender side, the progressive email services use Outlook's email client to generate emails. On the receiving end, we support both CoFi-enabled clients running Outlook and standard third-party email readers.

We implemented emails as CoFi shared documents that are written only by the email's sender but are read by one or more recipients. Our sender adaptation policy propagates the text content of new emails, transcodes images into a progressive JPEG representation and sends only portions of an image's data. The sender can propagate fidelity refinements for images by selecting the email from a special Outlook folder and re-sending it. The image fidelity refinements are available to CoFi-enabled recipients as soon as they reach the CoFi remote proxy. For CoFi-enabled recipients, our adaptation policy fetches the email's text content and transcoded versions of its image attachments. Readers request fidelity refinements by clicking a refresh button added on Outlook's toolbar. Finally, we support third-party email readers by composing a new email message once all images have reached full fidelity.

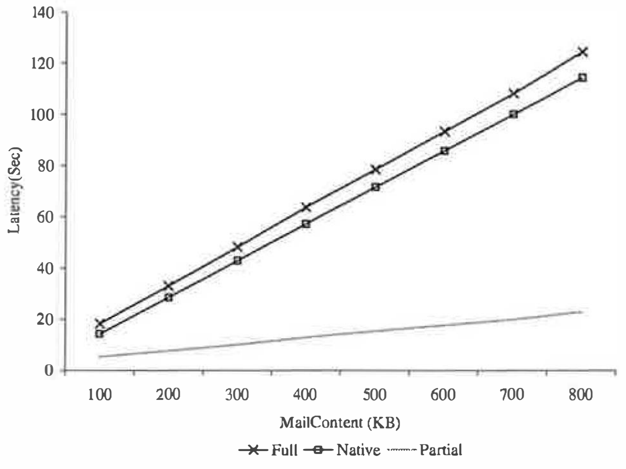


Figure 7: Latency for sending emails with and without progressive update propagation over 56 Kb/sec.

4.1.1 Progressive Update Propagation

Figure 7 plots the latencies for transmitting a set of synthetic emails consisting of a few text paragraphs and a variable number of image attachments each of size 100 KB over a 56 Kbps link. The plots show results for a run that uses Outlook without any adaptation support (*Native*), and two CoFi runs, one that sends the full images (*Full*), and a second that uses versioning to propagate partial-fidelity versions of the images (*Partial*). In this experiment, a partial-fidelity version correspond to the initial $1/7$ of the content of an image encoded in a progressive JPEG representation.

For the *Native* run, we measure only the time it takes to transmit the emails between the mobile client running Outlook and an SMTP server on the other end of the bandwidth-limited link. This accounts for the time that the mobile client has to wait before disconnection in order to propagate the email. We do not include the time it takes for the SMTP server to deliver the email to the recipients, as these operations can be done asynchronously and do not require the mobile client to remain connected. Similarly, for CoFi runs we measure only the time it takes to transmit the emails between the CoFi local and remote proxies and do not include the time needed to compose and send emails to third-party email recipients, or the time it takes for CoFi-enabled recipients to read the email adaptively.

Full demonstrates that the CoFi overhead is small, averaging less than 5% over all emails. In contrast, *Partial* shows that progressive propagation of the attachments reduces the latency by roughly 80%. The 5% overhead in CoFi corresponds to the cost involved in parsing the email content to find its structure, exchanging the control information, and transcoding the images.

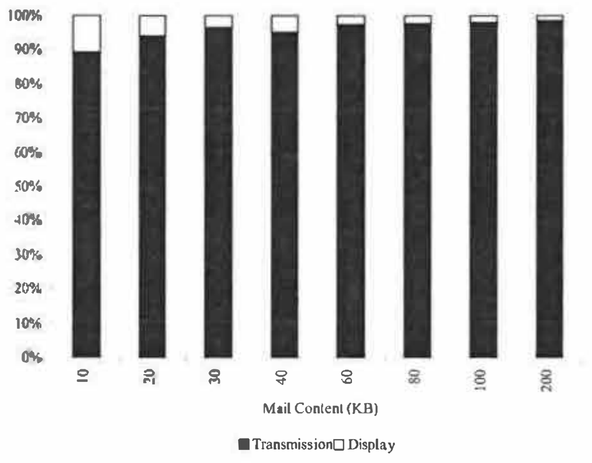


Figure 8: Breakdown of partial update propagation latency while sending emails.

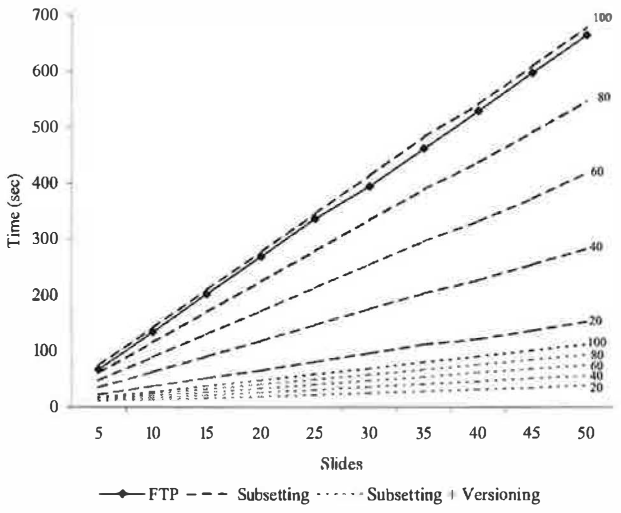


Figure 9: Latency for saving modifications to PowerPoint presentations with and without adaptation over 56 Kb/sec.

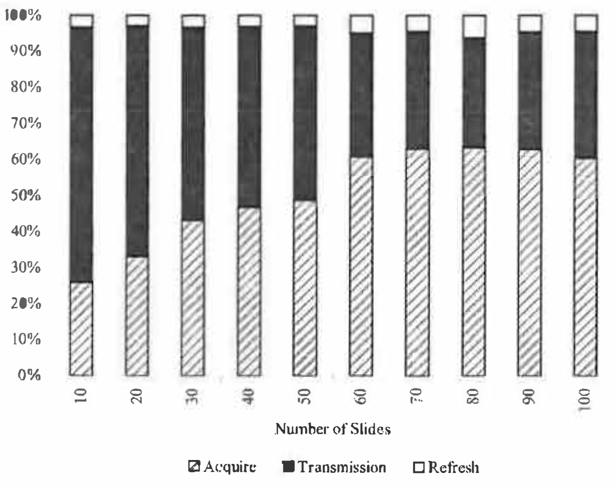


Figure 10: Latency breakdown for upgrading the fidelity of a single image in PowerPoint documents of various sizes.

4.1.2 Fidelity Upgrade

We measured the time it takes for a CoFi-enabled recipient to upgrade a partial-fidelity image to full fidelity. Figure 8 shows that the largest fraction of the time necessary for this fidelity upgrade is due to transmission (Transmission), and that only a small fraction of the time is spent on displaying the upgraded images in the application (Display). In other words, the overhead caused by CoFi's use of the API is very small. While we were not expecting the overhead to be significant, these results confirm that CoFi supports these kinds of adaptations in practice.

4.2 PowerPoint

We adapted PowerPoint to support adaptation-aware editing and progressive update propagation. Adaptation-aware editing reduces download time by enabling mobile clients to edit PowerPoint presentations that have been aggressively adapted. Previous work [5] demonstrated that loading text-only versions of PowerPoint presentations can reduce download latency over bandwidth-limited links by over 90% for large presentations. The rest of this section evaluates the benefits of progressive update propagation, quantifies the latency for updating a PowerPoint presentation with higher-fidelity data, and discusses our conflict resolution policy.

4.2.1 Progressive Update Propagation

CoFi-enabled PowerPoint propagates modifications progressively by saving back just subsets of the modified slides or embedded objects, or by transcoding embedded images into a progressive JPEG representation and saving just portions of the images' data. We implemented an adaptation policy that propagates modifications every time the user saves the document. The policy propagates the text content of any new or modified slides and transcoded versions of new or modified embedded images. Image fidelity refinements are then propagated on every subsequent save request until all images at the CoFi remote proxy reach full fidelity.

We evaluate the effectiveness of progressive update propagation by measuring the latency for saving modifications to a set of synthetic PowerPoint documents. We constructed our synthetic documents by replicating a single slide that contained 4 KB of text and a 80 KB image.

Figure 9 shows latency measurements for saving PowerPoint documents with up to 50 slides over a 56 Kb/sec network link. The figure shows latency results for transferring the documents over FTP and adaptation policies that use subsetting and versioning to reduce the data traffic. The FTP measurements give us a baseline for the time it takes to transfer the full document without any adaptation. We use this baseline to determine the effectiveness of our adaptation policies.

Figure 9 plots the results of 5 experiments that use subsetting to reduce latency. The numbers on the right hand side of the plot, next to each line, show the proportion of document slides that was saved back to the remote proxy. In this manner, the top most line corresponds to documents that were saved in their entirety, while the lowest subsetting line corresponds to documents where only modifications to 20% of the slides were saved. In all experiments, we assume that both the slide's text and single image were modified and had to be saved back. The top most subsetting line, which corresponds to saving the full document, shows that the CoFi overhead is small, averaging less than 5% over all documents. In contrast, all other subsetting experiments show significant reductions in upload latency. The last five lines in Figure 9 show the results for an adaptation policy that uses subsetting and versioning of images to further reduce upload latency. This policy converts images embedded in slides into a progressive JPEG representation and transfers only the initial 1/7 of the image's data; achieving even larger reductions in upload latency.

4.2.2 Fidelity Upgrade

Figure 10 shows the breakdown of the execution time for updating a single image with higher fidelity data. The figure shows that the API calls to replace the image (Display) account for a small portion of the overall latency, similar to what we saw for Outlook in Figure 8. More significantly, the figure shows that for large documents, roughly 60% of the time for upgrading the fidelity of an image is spent making sure the user did not modify the image we are about to update (Detect). Detecting modifications is time-consuming because PowerPoint's API does not support querying whether a component has been changed. Instead, we detect modifications by saving a temporary copy of the presentation on disk, parsing this copy, and comparing it with the local proxy copy of the presentation. Writing out a copy of the presentation to disk dominates the cost of all other factors in the total time taken for detecting modifications.

This experiment represents a worst-case scenario of having to write out the entire document to upgrade a single image. Under normal operation, we expect modification detection to benefit from PowerPoint's ability to write out modifications incrementally, as well as from the possible batching of multiple component upgrades into a single operation (i.e., updating a set of images at a time). Detecting modification is also only necessary if we allow editing of partial-fidelity images. If we disallow editing partial-fidelity images, then the cost to upgrade an image is just a few milliseconds over the time it takes to transmit the image over the bandwidth-limited link.

In any case, the large modification detection time results from specific limitations of the current PowerPoint API (which could be easily fixed by adding an API call for checking if a component has been changed), and is *not* a fundamental limitation of either adaptation-aware editing or progressive update propagation.

4.2.3 Conflict Resolution

We consider the following conflicts: one user modifies a slide while another user deletes it, two users move a slide to different positions in the presentation, or two users concurrently modify the same slide. We refer to these conflicts as *edit-delete*, *move-move*, and *edit-edit*, respectively. For simplicity, for the rest of this section, we refer to the copy of the presentation available at the remote and local proxies as the *remote* and *local* copies, respectively.

Our PowerPoint policy resolves *edit-delete* and *move-move* conflicts automatically. For *edit-delete* conflicts, our policy prioritizes editing over deletion, recreating the slide in the replica where it was deleted. For *move-move* conflicts, our policy gives priority to the local copy, moving the slide in the remote copy to reflect its position in the local copy. Finally, we resolve *edit-edit* conflicts by using the CoFi PowerPoint driver to present the two conflicting slides to the user, and prompting the user to resolve the conflict by either choosing one of the slides or by merging their content. The above policy is, however, just one of the possible ways to resolve conflicts, and we can easily envision variations or extensions to this simple policy.

The cost of conflict detection is highly dependent on the size of the documents. The bulk of the cost stems from writing out the document (see Section 4.2.2) and from transmitting the data over the network. All other aspects, including the execution of the conflict detection algorithm and the use of the APIs to display conflicts to the user, are insignificant.

5 Related Work

Support for partial propagation of modifications made to a shared database or file system has been provided before. This paper, however, is the first to introduce mechanisms that support propagating partial-fidelity versions of modifications, as well as their progressive improvement. WebDAV [30], and LBFS [20] implement file systems for wide-area and low-bandwidth networks. Coda [14], Ficus [25], and Bayou [29] provide support for document editing on disconnected devices. These systems differ from CoFi in that they are not aware of the fidelity level of the objects they replicate.

While various adaptation systems [1, 5, 7, 8, 11, 12, 17, 19, 21, 28] use subsetting and versioning to reduce document download time, CoFi is the first to provide adaptation support for multimedia authoring and collaborative work over bandwidth-limited devices.

Several efforts [3, 9, 18] have used component-based technologies to implement collaborative applications that adapt to variations on network connectivity, or have implemented collaborative applications that use the document's component structure to reduce conflicts or limit the amount of data that need to be present at the device [2, 6, 13, 22, 27]. These efforts, however, do not allow the propagation of partial-fidelity versions of modifications. MASSIVE-3 [10] uses transcoding to reduce data traffic necessary to keep users of a collaborative virtual world aware of each other. MASSIVE-3, however, implements a pessimistic single-writer consistency model.

6 Conclusions

We have described adaptation-aware editing and progressive update propagation, two novel mechanisms for supporting multimedia authoring and collaborative work on bandwidth-limited devices. Both mechanisms decompose documents into their components structures (e.g., pages, images, paragraphs, sounds) and keep track of consistency and fidelity at a component granularity. Adaptation-aware editing lowers download latencies by enabling users to edit adapted documents. Progressive update propagation shortens the propagation time of components created or modified at the bandwidth-limited device by transmitting subsets of the modified components or transcoded versions of the modifications.

We demonstrate that support for adaptation-aware editing and progressive update propagation can be added to optimistic and pessimistic replication protocols in an orthogonal fashion. Specifically, new states are added to the state machines that describe the replication protocols, but the existing states and transitions remain unaffected.

We have described the implementation of our CoFi prototype, which supports adaptation-aware editing and progressive update propagation for optimistic client-server replication. We have presented performance results for experiments with multimedia authoring and collaboration with two real world applications. For these applications, the ability to edit partially loaded documents and progressively propagate fidelity refinements of modifications substantially reduce upload and download latencies.

While the experiments in this paper focus on document-centric applications, the same principles can be extended to applications with real-time requirements, such as video or audio. Adaptation-aware editing could be used to support video editing, while progressive update propagation would be useful in situations where there is a benefit in retransmitting a higher-fidelity version of a video or audio stream, such as when a user listens to a recording multiple times.

References

[1] David Andersen, Deepak Basal, Dorothy Curtis, Srinivasan Srinivasan, and Hari Balakrishnan. System support for bandwidth management and content adaptation in Internet applications. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, California, October 2000.

[2] Rhonda Chambers, Dean Crockett, Greg Griffing, and Jehan-Francois Paris. A Java tool for collaborative editing over the Internet. In *Proceedings of the 1998 Energy Sources Technology Conference and Exhibition (ETCE '98)*, Houston, TX, February 1998.

[3] Keith Cheverst, Gordon Blair, Nigel Davies, and Adrian Friday. Supporting collaboration in mobile-aware groupware. *Personal Technologies*, 3(1):33–42, March 1999.

[4] Eyal de Lara, Dan S. Wallach, and Willy Zwaenepoel. Opportunities for bandwidth adaptation in Microsoft Office documents. In *Proceedings of the Fourth USENIX Windows Symposium*, Seattle, Washington, August 2000.

[5] Eyal de Lara, Dan S. Wallach, and Willy Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, San Francisco, California, March 2001.

[6] Dominique Decouchant, Vincent Quint, and Manuel Romero Salcedo. Structured cooperative authoring on the World Wide Web. In *Proceedings of the Fourth International World Wide Web Conference*, Boston, Massachusetts, December 1995.

[7] Christos Efstratio, Keith Cheverst, Nigel Davies, and Adrian Friday. Architectural requirements for the effective support of adaptive mobile applications. In *Proceedings of the Second International Conference on Mobile Data Management*, Hong Kong, January 2001.

[8] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. *SIGPLAN Notices*, 31(9):160–170, September 1996.

[9] Julian Gallop, Christopher Cooper, Ian Johnson, David Duce, Gordon Blair, Geoff Coulson, and Tom Fitzpatrick. Structuring for extensibility - adapting the past to fit the future. In *Proceedings of The*

CSCW2000 workshop on Component-based Groupware, Philadelphia, Pennsylvania, December 2000.

[10] Chris Greenhalgh, Jim Purbrick, and Dave Snowdon. Inside Massive-3: Flexible support for data consistency and world structuring. In *Proceedings of the Third International Conference on Collaborative Virtual Environments*, San Francisco, California, September 2000.

[11] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Building reliable mobile-aware applications using the Rover toolkit. In *Proceedings of the 2nd ACM International Conference on Mobile Computing and Networking (MobiCom '96)*, Rye, New York, November 1996.

[12] James J. Kistler and Mahadev Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[13] Michael Koch and Jurgen Koch. Application of frameworks in groupware - The Iris group editor environment. *ACM Computing Surveys*, 32(1es), March 2000.

[14] Puneet Kumar and Mahadev Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proceedings of the USENIX Winter 1995 Technical Conference*, New Orleans, Louisiana, January 1995.

[15] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[16] Yui-Wah Lee, Kwong-Sak Leung, and Mahadev Satyanarayanan. Operation-based update propagation in a mobile file system. In *Proceedings of the USENIX Annual Technical Conference*, Monterrey, California, June 1999.

[17] Baochun Li and Klara Nahrstedt. Qualprobes: Middlewate QoS profiling services for configuring adaptive applications. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, New York, New York, April 2000.

[18] Radu Litiu and Atul Prakash. Developing adaptive groupware applications using a mobile component framework. In *Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work (CSCW 2000)*, Philadelphia, Pennsylvania, December 2000.

[19] Lily B. Mummert, Maria R. Ebling, and Mahadev Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[20] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[21] Brian D. Noble, Mahadev Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. *Operating Systems Review (ACM)*, 51(5):276–287, December 1997.

[22] Francois Pacull, Alain Sandoz, and Andre Schiper. Duplex: A distributed collaborative editing environment in large scale. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW)*, pages 165–173, Chapel Hill, North Carolina, October 1994.

[23] Gerald J. Popek, Richard G. Guy, Thomas W. Page, Jr., and John S. Heidemann. Replication in Ficus distributed file systems. In *Proceedings of the Workshop on Management of Replicated Data*, pages 20--25, Houston, Texas, November 1990.

[24] Dave Ragget, Arnaud Le Hors, and Ian Jacobs, editors. *HTML 4.01 Specifications*. December 1999. `http://www.w3.org/TR/html4/`.

[25] Peter Reiher, John Heidemann, David Ratner, Gregory Skenner, and Gerald J. Popek. Resolving file conflicts in the Ficus file system. In *Proceedings of the Summer USENIX Conference*, pages 183–195, Boston, Massachusetts, June 1994.

[26] Luigi Rizzo. DummyNet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):13–41, January 1997.

[27] Martina Angela Sasse, Mark James Handley, and Shaw Cheng Chuang. Support for collaborative authoring via email: The MESSIE environment. In *Proceedings of 3rd European Conference on Computer Supported Cooperative Work*, pages 249–264, Milan, Italy, sep 1993.

[28] Mahadev Satyanarayanan, Jason Flinn, and Kevin R. Walker. Visual proxy: Exploiting OS customizations without application source code. *Operating Systems Review*, 33(3):14–18, July 1999.

[29] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 172–183, Cooper Mountain, Colorado, December 1995.

[30] E. James Whitehead and Yaron Y. Goland. WebDAV: A network protocol for remote collaborative authoring on the Web. In *Proceedings of the Sixth European Conference on Computer Supported Cooperative Work (ECSCW'99)*, pages 291–310, Copenhagen, Denmark, September 1999.

Characterizing Mobility and Network Usage in a Corporate Wireless Local-Area Network

Magdalena Balazinska
MIT Laboratory for Computer Science
mbalazin@lcs.mit.edu

Paul Castro
IBM T.J. Watson Research Center
pcastro@us.ibm.com

Abstract

Wireless local-area networks are becoming increasingly popular. They are commonplace on university campuses and inside corporations, and they have started to appear in public areas [17]. It is thus becoming increasingly important to understand user mobility patterns and network usage characteristics on wireless networks. Such an understanding would guide the design of applications geared toward mobile environments (e.g., pervasive computing applications), would help improve simulation tools by providing a more representative workload and better user mobility models, and could result in a more effective deployment of wireless network components.

Several studies have recently been performed on wireless university campus networks and public networks. In this paper, we complement previous research by presenting results from a four week trace collected in a large corporate environment. We study user mobility patterns and introduce new metrics to model user mobility. We also analyze user and load distribution across access points. We compare our results with those from previous studies to extract and explain several network usage and mobility characteristics.

We find that average user transfer-rates follow a power law. Load is unevenly distributed across access points and is influenced more by which users are present than by the number of users. We model user mobility with *persistence* and *prevalence*. Persistence reflects session durations whereas prevalence reflects the frequency with which users visit various locations. We find that the probability distributions of both measures follow power laws.

1 Introduction

Several recent studies characterize the usage of various wireless networks [3, 8, 9, 10, 15, 16]. Tang and Baker [15] focused on a university building and traced the activity of 74 users over 12 weeks. Kotz and Essien [8, 9] studied a university campus network with 1706 users scattered through 161 buildings with a total of 476 access points. Balachandran et al. [3] examined usage of a wireless network in a large auditorium during a three day conference. Tang and Baker [16] also studied the Metricom metropolitan-area packet radio wireless network, a public network with approximately 25,000 radios. Lai et al. [10] analyzed a combined wireless and wired network, but that study was limited to only eight users.

Each study presents patterns of user mobility and network usage characteristics for one particular domain. In this paper, we complement these studies by presenting results from a four week trace gathered on a corporate wireless local-area network (WLAN). Our trace presents the activity of 1366 users. We use our trace as well as results from previous research to extract common characteristics of WLAN usage and to highlight and explain usage differences. We focus on population characteristics, load distribution across access points (APs), user level of activity, and user mobility.

We find that variations in the number of wireless users over time closely follow patterns of the underlying population, even though most users access the wireless network a fraction of days and a fraction of time. Hence, the number of users on a network might be adequately modeled by scaling down general population models.

Our study shows that there exist large personal differences in users' mobility as well as in their data transfer rates. Some users transfer over 1Mbps on average while others transfer less than 10Kbps on average. In general, we find that user average transfer rates follow a power law. The aggregate data transfer rate seen by an access point does not seem to depend on the number of users associated with the access point, but rather on which users are present. In each building, approximately 30% of access points owe over 40% of their load to the most active 10% of users on the network. Location of an access point also plays a role in the aggregate load it observes.

Users spend a large fraction of their time and long periods of time at a single location, which we call their *home* location. Interestingly, they do not reduce their network usage when moving away from that location and changing location more frequently. We model user mobility with *persistence* and *prevalence*. Persistence measures how long users stay continuously associated with the same access point and prevalence reflects how frequently users visit various locations. Our definitions are based on Paxson's definitions of "routing persistence" and "routing prevalence" in his study of Internet routing stability [13]. We find that the probability distributions of both measures follow power laws. We use prevalence metrics to classify users into different mobility categories. We find that 50% to 80% of users are occasionally or somewhat mobile: they spend most of their time at a single location, but periodically visit other locations.

Comparing our results with other studies, we find many similarities in mobility and network usage characteristics. We find that these characteristics are best explained by factors orthogonal to whether the network runs on a campus, in a corporation, or in a public environment. The main factors influencing network usage include personal differences between users and function of various locations (including scheduled events). The main differences in user mobility appear among locations serving as primary places of work and locations visited occasionally. The density of resources (classrooms, conference rooms), and differences between individual users also influence mobility significantly.

The rest of the paper is organized as follows. We first present the methodology used to gather our trace in Section 2. In Section 3, we describe the characteristics of our user population and contrast it with the population in previous studies. In Section 4, we describe load distribution across access points and analyze factors influencing access point load. Section 5 presents and compares user mobility characteristics in each environment. We also introduce metrics for describing these characteristics. In Section 6, we discuss how some of our findings may benefit network deployment, application design, and simulation of user mobility. We conclude in Section 7.

2 Methodology

The 802.11b wireless local-area network that we studied is spread throughout three large corporate buildings hosting computer science and electrical engineering research groups. The largest of the buildings, which we call LBldg, has 131 access points and is approximately 10 miles away from the other buildings. The other buildings, MBldg and SBldg, are adjacent to each other. They have 36 and 10 access points respectively. The placement of access points in buildings is based on geometry (one access points per corridor, for instance). Extra access points are placed in a few highly used rooms, such as a customer laboratory in SBldg.

The network is configured to run in infrastructure mode, in which wireless clients connect to the wired network through access points distributed in the environment. All 177 access points are Cisco Aironet 350s. We observed a total of 1366 unique MAC addresses. Laptops were by far the predominant devices on the network. We do not have information whether any other types of devices were used at all. We assume that each unique MAC address corresponds to a user, even though it is possible for a single user to have more than one MAC address or for users to trade cards with each other.

We used SNMP [4] to poll access points every 5 minutes, from Saturday, July 20th 2002 through Sunday, August 17th 2002. We chose 5 min intervals to ensure that our study would not affect access point performance. We collected information about the traffic going through each access point as well as about the list of users associated with each access point. For each user, we retrieved detailed information on the amount of data (bytes and packets) transferred, the error rates, the latest signal strength, and the latest signal quality. We polled all access points except three located in MBldg that did not respond to SNMP requests.

Due to a power failure, there is a one-hour hole in the data (07/30/2002 from 1pm to 2pm). For unknown reasons, we also have a few holes in the data gathered at a few of the access points during the evening and night of 08/08/2002. Due to periods where access points were heavily loaded, some sample intervals stretch to 10 min.

Users were not informed that the study was performed. The only sensitive information that we gathered were the MAC and IP addresses of network cards, as well as the names assigned to access points. To ensure user privacy, we anonymized all three types of information. We did not map access points to explicit locations or track individual users. We only present aggregate results.

All data from our trace is available for download at the following location: `http://nms.lcs.mit.edu/~mbalazin/wireless/`.

3 Wireless user population characteristics

We saw a total of 1366 distinct users in our four week trace: 796 users spent most of their time in LBldg building, 437 in MBldg, and 133 in SBldg. Figure 1 shows the total number of users present on the network every day of the trace. Figure 2 shows the number of users present on the network during different hours of a day on weekdays. We show the 10th, 50th, and 90th percentile values registered for each hour throughout the trace. For all three buildings, the patterns reflect the office environment and normal office work hours. We also note a slight reduction in the number of users around lunch time (Figure 2). However, since the reduction is small, we conclude that most users work through lunch or leave their laptops on

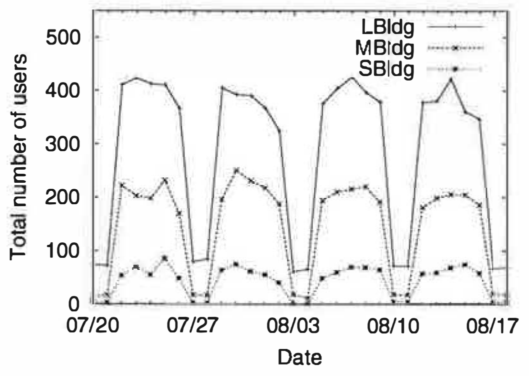


Figure 1: Total number of wireless users in each building on each day of the trace. The trace starts on Saturday, July 20th 2002. The figure show patterns of a normal office work week.

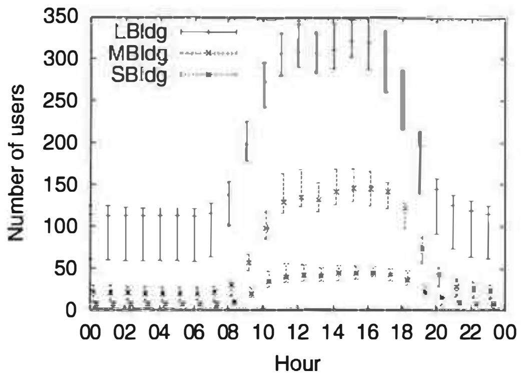


Figure 2: Number of users per hour on weekdays. For each building, the 10th, 50th, and 90th percentile values for each hour are shown. The figure shows a strong pattern of regular office work day.

while they eat, so the machine is "present" even if there is no activity. Also, some users stay late at night or leave their laptops on when they go home, since the number of wireless users is greater than zero during the night.

These patterns are similar to those found at university campus locations used for working (offices, libraries, academic buildings) [8, 9, 15]. They differ from on-campus locations such as dormitories [8, 9] and public metropolitan networks [16], which users access both from work and from home. In these environments, peaks in the number of users appear during evening hours. There are also much lower reductions in the number of users on weekends. Small scale networks such as a single conference room [3] show much more variability in the number of users due to the impact of scheduled activities.

Therefore, *daily and hourly patterns in numbers of wireless users on a network are closely tied to patterns in the underlying population*. Differences appear not so much among public, academic, or corporate networks but among networks that cover usage at the work place, at home, or during a specific event.

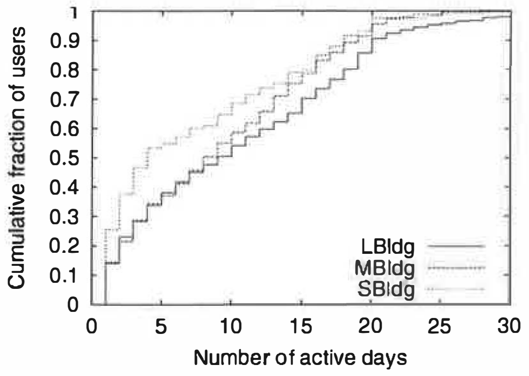


Figure 3: Number of days users are present in the trace. The distribution is uneven at the edges. A large fraction of users (around 22% to 38%) appear only one or two days in the trace. Less than 10% of users come more often than the 20 work days of the trace and a fraction of these are laptops left in offices.

To further characterize the user population, Figure 3 presents the cumulative distribution of the number of days each user appeared in the trace. Each user was counted only in the distribution of the building where the user spent most of his or her time. The number of days that users are present varies greatly: only 12% to 25% of users are present more than 18 out of the 20 work days, whereas 22% to 38% of users appear only during one or two days. We suspect that the latter group are outside visitors mostly from other sites that the company has in the same metropolitan area. It is interesting to note that the *great variety in the number of days that users are present does not influence the regular pattern* shown in Figure 1. The presence of visitors and the absence of employees must therefore be uniformly distributed.

In terms of the fraction of days that users access the network, our distribution is similar to a single building on a university campus [15]. Compared with a whole campus [8, 9] our trace has more users appearing only one or two days (visitors) and fewer users appearing more than 2/3 of the days. The higher uniformity of a campus wide distribution might be related to the fact that the study tracks many users for prolonged periods of time (i.e., students living on campus) and not only when they come to work in specific buildings.

Additionally, we computed the fraction of time users remain on the wireless network on days where they actually use it. We found that 50% of users remain connected 60% to 100% of the work day.

4 Load distribution across access points

In this section we examine load distribution across access points. According to current guidelines [1, 5], access

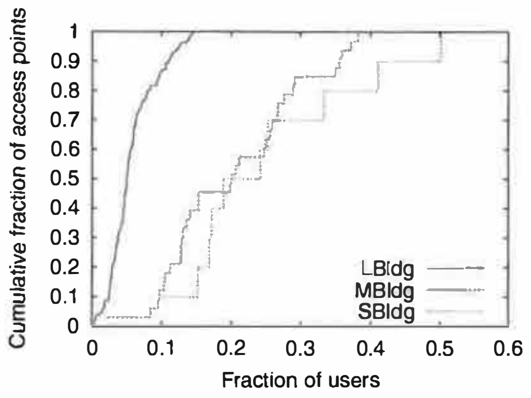


Figure 4: Fraction of users seen at each access point throughout the whole trace. The fraction is computed with respect to users who visited each building. The figure shows a wide disparity across access points and across buildings.

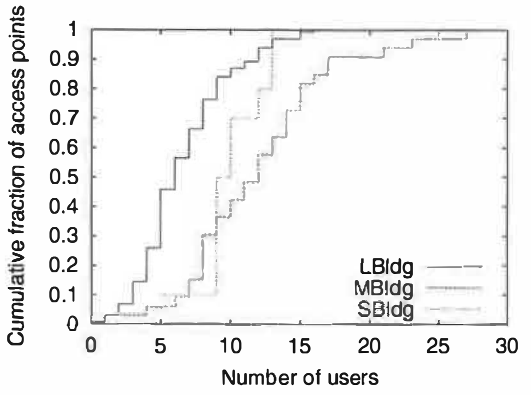


Figure 5: Maximum number of users ever simultaneously associated with each access point. The figure shows a wide disparity across access points.

points should be distributed based on the physical aspects of buildings, the signal strength, and signal-to-noise ratios, as well as the number of users and their application mix. In this section, we examine how load is balanced across access points in real settings. We examine the user distribution, the total amount of data transferred, as well as the data transfer rates. We also examine factors influencing access point load.

4.1 Users

Figure 4 shows the fraction of users seen at each access point throughout the trace. A few access points see a small fraction of users: 10% of access points in LBldg see only 2.5% of all users who visited the building. Others see a greater fraction of users, some as much as 50%. Differences between buildings are partly explained by building sizes (LBldg is much larger than the other two) and numbers of access points: LBldg has 131 access points, MBldg has 36, and SBldg has only 10. Since approximately twice as many users visited LBldg as did MBldg, LBldg has a much smaller ratio of users to access points (around 7 versus 16 and 30 for MBldg and SBldg, respectively).

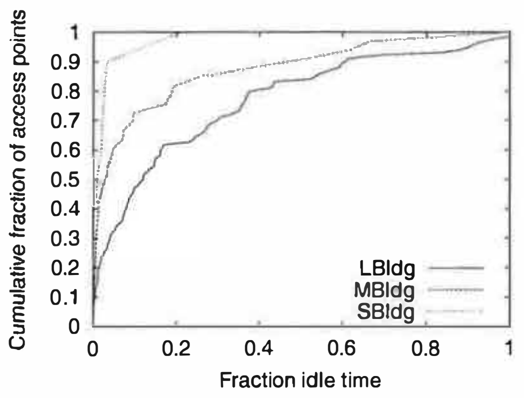


Figure 6: Fraction of time that access points are idle during a normal work week (Monday-Friday, 9am to 6pm). Most access points are used almost all the time. However, a few access points are idle large fractions of time. The fraction of time was computed as the fraction of samples without any users.

Figure 5 shows the maximum number of users simultaneously associated with each access point. Some access points see few simultaneous users: 40% of access points never see more than 10 users, while other access points see as many as 30 simultaneous users. Some of the access points with the highest numbers of simultaneously associated users correspond to large auditoriums and cafeterias.

University campuses [8, 9, 15] and large-scale public networks [16] also see great disparity in the average and maximum number of users handled by access points. This is not surprising as these values depend on the popularity of certain locations (auditoriums or cafeterias for example). Hence, except for small-scale networks [3], *popularity differences appear in all environment studied.* On a small scale, Balachandran et al. [3] find that users are distributed rather evenly across access points.

Given the regular work schedule of our corporate population, access points are idle (no user is associated with them) a large fraction of the time when weekends and nights are considered. However, access points are also idle some fraction of the time during normal working hours. Figure 6 shows the fraction of idle time for cumulative fractions of access points in each of the three buildings. Most access points are used almost all the time during a work week. For both MBldg and SBldg, 75% of access points are idle less than 10% of the time. However, 10% of the access points in LBldg are still idle over 75% of the time. On the university campus that Kotz and Essien study [8, 9], over one third of access points are

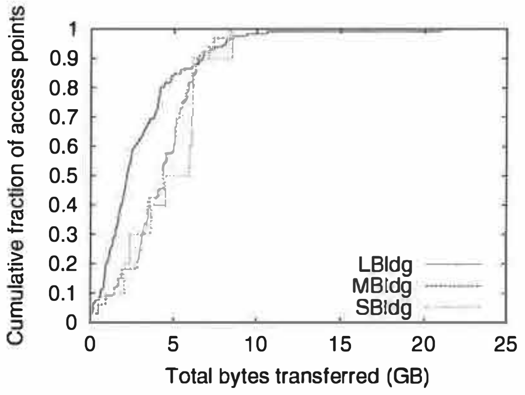


Figure 7: CDF of the amount of data transferred from each access point to wireless users over the duration of the trace. The figure shows an uneven usage of resources. Data from August 7th was ignored in this computation as all access-point counters were reset by a third party around 19:40 that day.

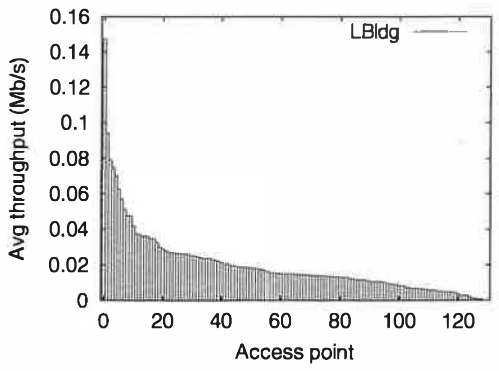
idle on a typical day. Hence, *in all environments, good coverage requires deploying resources, even in locations where they are seldom used.*

4.2 Data transferred

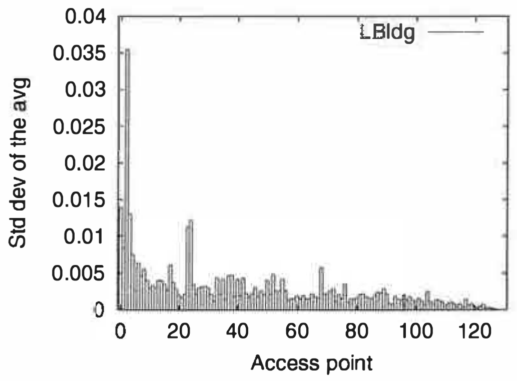
Figure 7 shows the total amount of data forwarded through each access point during the trace. Access points are ordered by increasing amount of data they forwarded, and the cumulative fraction of access points is indicated for each amount. Due to space limitations, we only present results for traffic going from access points to wireless users. The graph for the opposite direction is similar, though with slightly lower values. The amount of data forwarded varies considerably across access points (from close to 0 up to 21GB[1]), indicating an uneven usage of resources.

Figure 8 shows the average throughput of each access point. We define access point *throughput or load as the total amount of bytes an access point forwards for any associated user in either direction over a given period of time.* We computed these averages in two steps. For each access point, we first computed the average throughput for each sampling interval (interval between two consecutive polls of the same access point). We then computed the mean of these values. Figure 8 shows that much more data is transferred on average at some access points than others (with small errors on these averages, as shown in Figure 8(b)). We obtain similar graphs for the other two buildings. In the following sections, we discuss these throughput differences and examine factors that influence access point load.

[1]Throughout the text, $1MB = 2^{20}Bytes$ and $1Mbps = 2^{-3}MBps$



(a) Average access point throughput



(b) Standard error on average

Figure 8: Average throughput for access points in LBldg. Only intervals with users were considered in averages. Access points are ordered by decreasing average.

4.2.1 Correlation between number of users and load

Figure 9 shows access point throughput for various numbers of associated users for MBldg. Throughputs were computed over individual polling intervals. The figure shows that little correlation exists between the number of users and access point throughput. For more than 14 users, the outliers (99th percentile) seem to decrease, but this is due to the smaller number of samples with so many users. We show the results for MBldg as access points saw highest numbers of simultaneously associated users most frequently. The graphs for the other buildings show even less correlation.

We computed the correlation coefficients for each building, and found 0.10, 0.20, and 0.15 for LBldg, MBldg, and SBldg respectively. For intervals where access point load exceeded 100Kbps, the number of users and the load are even less correlated (−0.14, 0.03, and −0.06). This phenomenon can be explained by noticing

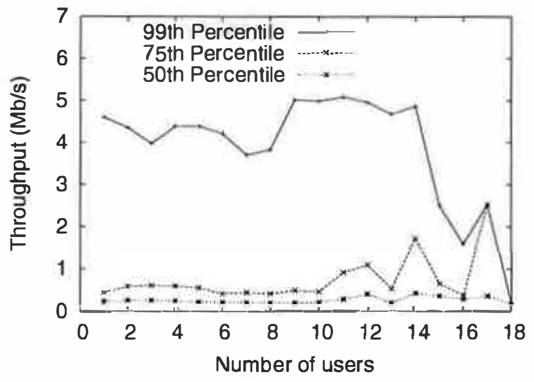


Figure 9: Throughputs measured at access points in MBldg for various numbers of associated users. 50th, 75th, and 99th percentile shown (only samples over 100Kbps were taken into account). The figure shows that little correlation exists between the two numbers.

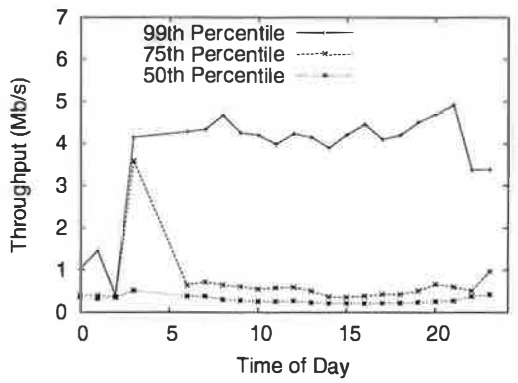


Figure 10: Throughputs (per polling interval) measured at every access points on each hour of the day on every workday in LBldg: 50th, 75th, and 99th percentile (of samples with values over 100Kbps) shown. No correlation seems to appear between throughput and hour of the day.

that most users are passive most of the time. When only passive users are present, increasing their number slightly increases the load. However, as soon as some users become active (i.e., start transferring large amounts of data), they drastically increase the average throughput and the influence of other users becomes insignificant.

We located a few access points that had the higher average transfer rates. They were our "dining conference room," laboratories, and conference rooms serving small meetings and teleconferences. These locations differ from most popular and crowded locations corresponding to cafeterias and auditoriums.

Tang and Baker [15] find that: "Usually, the throughput as a whole increases as the number of users increases" (i.e., the total throughput through routers increases with the number of users). However, they find that: "the maximum throughput is achieved by a single user and application." Kotz and Essien [8, 9] also notice little correlation between the number of users and the amount of traffic going through access points. They find the largest numbers of users at access points located in lecture halls, while most traffic comes from residences. However, it is not clear from their results whether the difference is attributable to the extra amount of time that users spend daily on the network at residences or their level of activity at these locations. Balachandran et al. [3] point towards the fact that: "load distribution [...] does not directly correlate to the number of users at an access point." Indeed, they find that peak load is not achieved when the maximum number of users are present. They also find that although the number of users is almost constant, load varies considerably over time. We confirm their conclusion on a larger scale. Additionally, Figure 9 shows that the number of users and the load are rather uncorrelated regardless of whether the load is high or not.

We conclude that *offered load and number of users associated with an access points are weakly dependent* in our environment, but also in the other environments studied.

4.2.2 Correlation between time of day and load

Figure 10 shows access point throughputs registered on various hours of the day. Throughputs were computed separately for each access point and each polling interval. The value was then associated with the hour of the first poll. The figure shows that little correlation exists between time and throughputs other than the fact that sometimes users are not present on the network, as shown earlier on Figure 2. The correlations coefficients between time of day and load are 0.016, 0.020, and 0.030 for LBldg, MBldg, and SBldg respectively. Similarly, in the trace presented by Balachandran et al. [3], the offered load oscillated between 0Mbps and 2Mbps as long as users were present on the network (during morning and afternoon sessions), showing little correlation between load and time of day.

Interestingly, in MBldg, we found that the few users who stay later at night generate only little activity whereas in LBldg, a lot of activity persists on the network until midnight (as shown on Figure 10). This shows a difference in the characteristics of the wireless population that did not appear when examining only the number of users on the network (Figure 2). Also, for all buildings, users detected around 3am to 6am were idle laptops left on in offices, since no activity was detected during these periods.

Even though load on access points is not directly correlated with the time of day, it may be influenced by specific events such as regularly scheduled meetings. For instance, for one of the access points, most peaks (above 2Mbps) occurred regularly between 12pm and

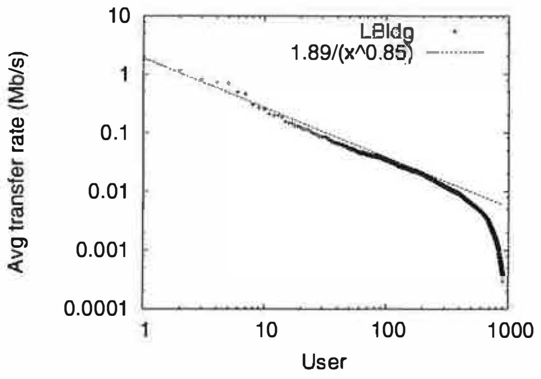


Figure 11: Distribution of average individual user transfer rates in LBldg. Averages follow a power law, except for passive users with average transfer rates under 10Kbps.

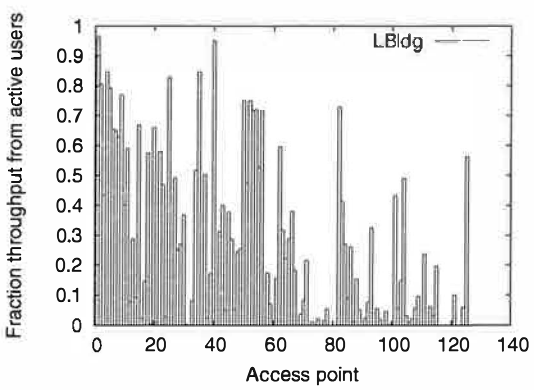


Figure 12: In LBldg, fraction of total throughput attributable to users with average transfer rates higher than 0.04Mbps. Access point indices follow those in Figure 8 (decreasing average throughput).

2pm. However, only few access points showed such clear patterns.

4.2.3 Personal user differences

Personal user differences are another factor influencing access point load: Some users are more active than others. To appreciate these differences, we compare the average rates at which users transfer data (in either direction). For any two consecutive polls, we compute the average transfer rate of each user who remained associated with the same access point over the interval. We then computed the average of all values for each user to get the overall average for that user.

We compare individual average traffic rates to determine personal differences in user's level of activity. Figure 11 shows a great disparity among users. Some users have quite high average traffic rates while others are hardly ever active. Except for passive users (transfer rate under 10Kbps), the *average user transfer rates follow a power law distribution.* For both MBldg and SBldg, we find a similarly shaped distribution with a 10Kbps threshold under which the distribution does not follow a power law anymore.

Whenever users with high network usage characteristics appear, an access point may expect its load to increase significantly. Figure 12 shows the fraction of the throughput due to users whose average data transfer rates are above 0.04Mbps. Access point indices correspond to those of Figure 8 (i.e., decreasing average access point throughput). These users represent only 10% of all users however they account for over 40% of the bandwidth usage at over 30% of access points. We obtain similar graphs for the other buildings. We also looked at users whose average rate was above 0.02Mbps. They represented 20% of all users, but they were responsible for over 40% of load at 60% of all access points.

Tang and Baker [15] find some differences among what they call user sub-communities (users whose offices are grouped around different access points). They find that, in general: "While the wings with most users (2b and 3b) also have the highest peak throughput, the users on the 3b wing attain that throughput more often." Other than this general finding, few conclusions are drawn in the other studies about personal user differences. We find that *these differences are significant.* Since our trace monitored 1366 users as they worked in offices, attended meetings, and relaxed in common areas, we believe that personal differences will extend to other environments as well.

4.2.4 Influence of access point location on load

Figure 12 shows that access points with high average throughputs (those at lower indices in the figure) owe slightly larger fractions of their throughputs to the most active users. Active users also use a large fraction of bandwidth at locations with little total activity. Therefore, the fact that some access points have much higher average transfer rates than others is also due to other factors than which users are present. In this section, we analyze the impact of access point location on its load by testing whether location influences users' level of activity.

To determine whether user's level of activity depends on location or not, we perform a one-way classification analysis of variance [6]. The factor that we are investigating is location. The hypothesis is that there is no effect of location on a user's data transfer rate. For each user, this analysis method compares the distribution of the rates achieved at each location visited (F-test [6]).

We only examine users who visited at least two locations and whose average transfer rate was above 10Kbps. For LBldg the classification is statistically significant for 27% of users. We reject the hypothesis of location neutrality in all these cases, and conclude that, for these users, location significantly affects transfer rate. Similarly, we obtain a significant result for 32% of users in MBldg and 23% of users in SBldg. We reject the hypothesis for all of them. Hence we conclude that *user transfer rates are affected by the location.*

From anecdotal evidence, we know that, in our corporation, during talks held in large auditoriums, users mostly check their email or browse the Web. During conference calls held in small rooms, users go over presentations and download attachments pertaining to the meeting, hence using much more bandwidth. We plan to investigate the relationships between applications and location in future work.

Other studies also seem to find some relationship between location and activity level, mostly because location determines the type of activities that users pursue. Balachandran et al. [3] never see peaks greater than 0.57Mbps for any user. Kotz and Essien [8, 9] find that the daily throughput per MAC address varies greatly between buildings, with residences seeing much more traffic than social locations. They do not indicate whether the differences are attributable to higher throughputs or to the length of time that users spend in each location. However, they do find differences in the types of applications predominantly used in each campus building.

4.2.5 Access point peak throughput periods

In our data set, we observed many polling intervals where average access point throughputs exceeded 3Mbps. We observed extremely few intervals with 5Mbps or higher averages. Therefore, in this section, we present results for intervals averaging over 4Mbps. We call such intervals *peak throughput periods* or *peaks*. We find that peaks last short periods of time and seem highly correlated with location. However, the network we studied was well provisioned and did not seem to experience much overload.

Figure 13 shows the number of polling intervals (consecutive or not) where the average transfer rate exceeded 4Mbps for each of the 131 access points in LBldg. In the figure, access points are ordered by decreasing number of peaks. Some access points experience peak throughput periods quite often while the transfer rate at others never exceeds 4Mbps. A few of these peaks lasted over an hour, but 48% of them lasted only for one polling interval in both LBldg and MBldg. The fraction was 64% in SBldg. In [15], Tang and Baker found that in their network, throughputs greater than 3Mbps were due mostly to a single user rather than distributed across several users. They also found that some locations were seeing significantly more peaks than others.

We also looked at how often more than one access point experienced a period of high load. For LBldg,

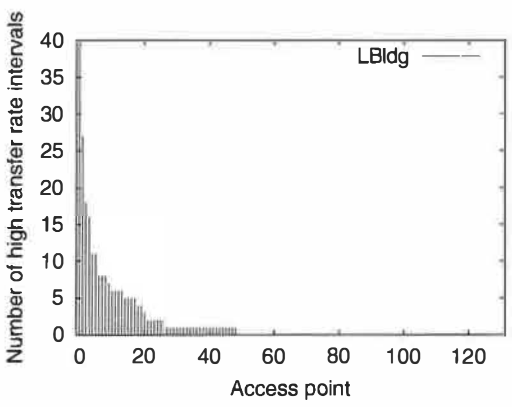


Figure 13: Distribution of the number of high data transfer rate intervals across access points. Some access points see high transfer rates quite often while most access points never see high transfer rates (access points are ordered by decreasing number of peaks).

there were 188 events where load exceeded 3.5Mbps at some access point. 50 of these events happened at the same time or within a few minutes of one or two other events. However, since we do not track access point location, some of these simultaneous events are probably unrelated. Hence, most peaks affect a single access point at the time.

We conclude that individual user differences have a high impact on access point load: 20% of users account for 40% of the data transferred at over 60% of the access points. Access point location also influences load, but much less. Popular locations see many simultaneous users and a larger fraction of all users, but the number of users does not influence load significantly. Unpopular locations remain idle most of the time. Time of day influences the number of users present on the network, but it does not influence access point load significantly, even during the night. Finally, many network usage characteristics such as the uneven distribution of users and load across access points, and the differences in location popularity, are independent of whether the network is deployed on a university campus, a corporation or at a conference.

5 User mobility characteristics

In this section, we examine user mobility characteristics and compare our results with the characteristics found in the other studies. We assigned a *home building* to each user corresponding to the building where they spent most of their time.

For each building, Table 1 shows the fraction of users who visited only that building, either of the other buildings, or all buildings. Most wireless users stay within one

	LBldg	MBldg	SBldg
one bldg	81%	55%	72%
two bldgs	16%	34%	26%
three bldgs	3%	11%	2%

Table 1: Fraction of users who visited one, two or all three buildings. Each user is counted with the building where the user spent most of his or her time.

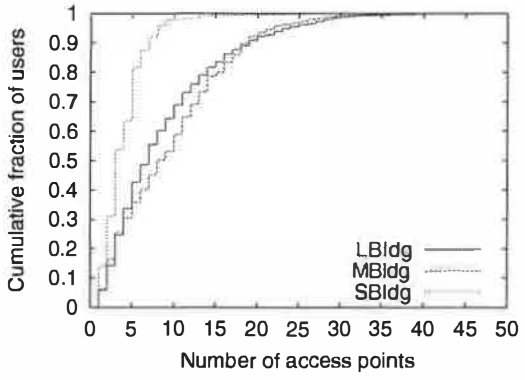


Figure 14: Number of access points visited by users during the whole trace.

building, but a significant fraction (20% to 45%) move between two or more buildings. This is mostly the case for MBldg and SBldg, located near each other. Only a small fraction (up to 11%) of users visit all three buildings. These mobility patterns are much more constrained than those found on a university campus [8, 9] where the median user in their trace visited five buildings. The difference is due to a higher concentration of resources (libraries, conference rooms) within each of our corporate buildings. Also, many university users both work and live on campus and visit different campus locations for each type of activity.

Figure 14 shows the number of access points each user visited during the trace. The figure shows cumulative fractions of users who visited increasing numbers of access point. We find that 50% of users in SBldg, MBldg, and LBldg visited respectively 3, 9, and 7 access points or fewer. These numbers include users who use their wireless cards only a few days in the trace. On the other hand, 50% of users visit between 7 and 40 access points, with one user visiting as many as 50. University campus users show even greater mobility disparities than corporate users: the median corporate user visits a similar number of access points as the median university user [8, 9], but the tails of our distributions (Figure 14) are shorter. Hence, there might exist small differences between user populations, but the differences might also be due to the fact that Kotz and Essien [8, 9] study many users while attending activities other than working.

Figure 15 shows the distribution of the number of

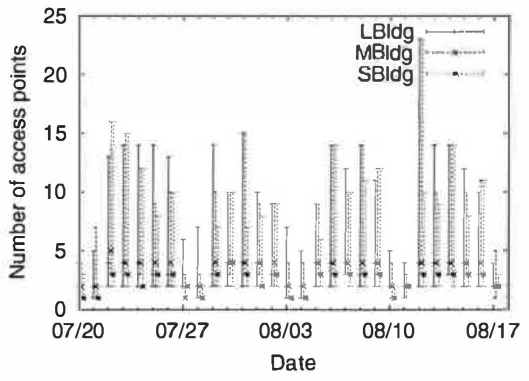


Figure 15: Distribution of number of access points visited by users each day of the trace. The 50th, 75th, and 99th percentiles are represented. Note that graphs for MBldg and SBldg are slightly offset along the x-axis to improve readability.

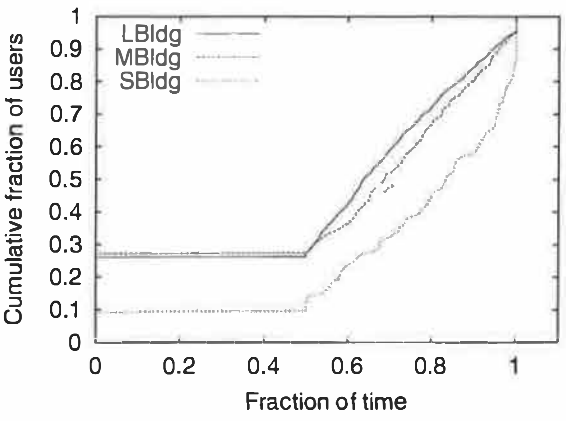


Figure 16: Fraction of time that users spend at their home locations. The fraction of time was measured as the fraction of samples where the user was seen at its home location.

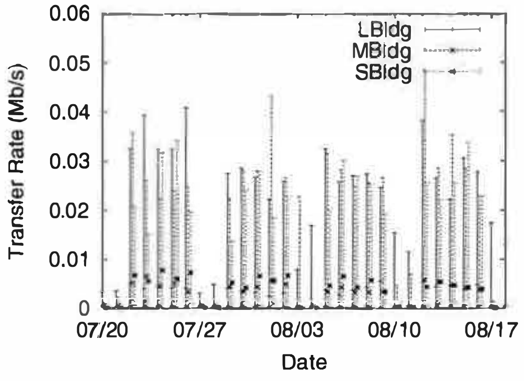
access points visited daily (50th, 75th, and 99th percentiles). The daily values are lower than the cumulative values presented in Figure 14, showing that users visit different locations on different days. On a daily basis, the least-mobile users (up to the 50th percentile) visit up to three access points in a single day. Most users (up to 75% of them) visit up to 3 or 5 access points, and the most-mobile 25% of users visit between 5 and 25 access points in a single day. As in our study, Tang and Baker [15] also find differences in user mobility, categorizing many as stationary, some as somewhat mobile, and a small fraction as highly mobile. Hence, in different environments there exist large *personal differences in user mobility, with most users spending a large fraction of their time at one location.*

5.1 Home location and guest location

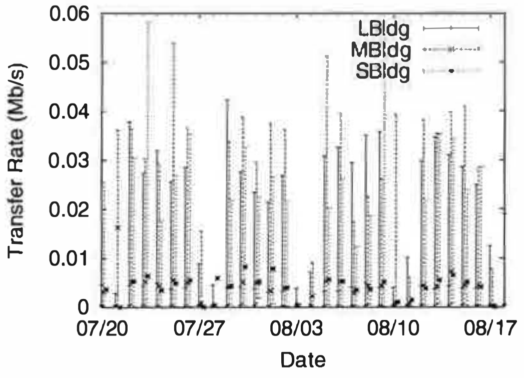
Since most users are stationary a large fraction of the time, we compare user behavior in the location where they spend most of their time — their *home location* — with their behavior in other locations — *guest locations.*

To determine each user's home location, we could simply identify the access point with which they are most frequently associated. However, visitors or users who use the Ethernet when working in their offices should not be assigned a home location. Therefore, we fix a threshold on the fraction of time that a user must spend with an access point for it to be considered the user's home location. We computed home locations for thresholds of 30%, 40% and 50%. With 30% or 40% thresholds, a few users who divided their time rather equally among various buildings ended-up with a home location in the wrong building. With a 50% threshold, 10% to 25% of users did not have a home location, but all home locations were within buildings where users spent most of their time. We therefore chose to use a 50% threshold to find user home locations. Figure 16 shows the cumulative distribution of the fraction of time spent by users at their home location. Users spend up to 100% of their time at their home location, with half the users spending at least 60% of their time there. Given the low daily user mobility found in the other studies, we expect users in other environments to show similar distributions of the amount of time spent at a single home location.

We find that the median user transfers around 20 MB in a single day at his or her home location which is similar to the amounts of data transferred by university users [8, 9]. At guest locations, users transfer approximately half that amount. However, since, by definition, users spend a large fraction of time at home locations, time plays a leading role in this difference. Figure 17 shows average daily transfer rates for home and for guest users (computed as the total amount of data transferred by a user divided by the total amount of time the user spent associated with home or guest access points during that day). Median values are similar for both guest and home users, but outliers (90th percentile values) show higher activity at guest locations. *Hence, mobility does not seem to have a negative impact on user transfer rates.*



(a) Transfer rates at home locations



(b) Transfer rates at guest locations

Figure 17: Average daily data transfer rates per user, at home and at guest locations. The 10th, 50th, and 90th percentiles are represented.

5.2 Prevalence

To better model the mobility of a user population or that of an individual user within the population, we compute two metrics: access-point *prevalence* in user traces and user *persistence* at various locations. These notions are motivated by Paxson's analogous definitions [13]. These two metrics characterize mobility patterns independently of the duration of the trace and independently of the amount of time that users spend on the network. We start by presenting prevalence metrics. We discuss persistence in the following section.

Access-point prevalence in a user's trace is the measure of the fraction of time that a user spends with a given access point. If a user visits an access point frequently or spends a lot of time at the access point, the prevalence of this access point in the user's trace will be high. Home locations therefore have high prevalence values whereas guest locations have lower prevalence values.

The prevalence distribution for a network is a matrix where each row corresponds to an access point and each column corresponds to a user, as illustrated in Figure 18. We compute one prevalence matrix for each building to compare mobility within each building. Figure 19 shows the probability distribution of prevalence values from the LBldg matrix. Zero-value prevalences have been discarded from the graph as most users visit only a few access point (so zero-value prevalences domi-

	U_1	U_2	...	U_n
AP_1	$Prev_{1,1}$	$Prev_{2,1}$	...	$Prev_{n,1}$
Ap_2	$Prev_{1,2}$	$Prev_{2,2}$	...	$Prev_{n,2}$
...	...	...	...	
AP_k	$Prev_{1,k}$	$Prev_{2,k}$	...	$Prev_{n,k}$

Figure 18: Prevalence matrix for a network of n users and k access points.

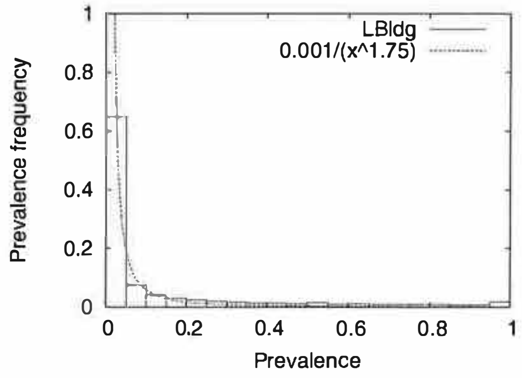
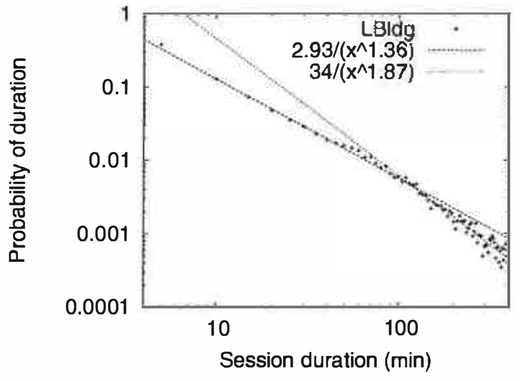


Figure 19: Probability distribution of prevalence values for LBldg. Zero valued prevalences are not counted. The other values are in bins of size 5%. The distribution follows a power law with a low exponent.

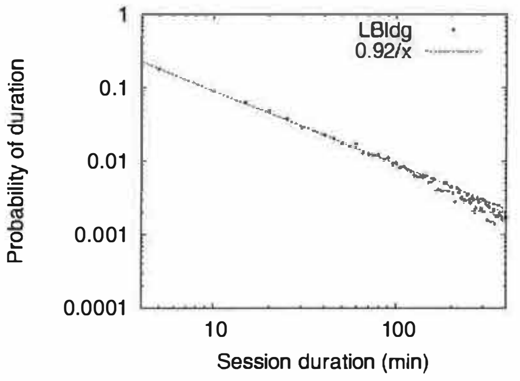
nate). The graph shows that users visit a few access points frequently (prevalences higher than 50% have non-zero probabilities) while visiting most access points rarely (prevalences of 0% to 5% are frequent). More precisely, the prevalence probability distribution follows a power law with a low exponent as shown in the figure. We obtain almost identical graphs for MBldg (with the same power law). For SBldg, we find a significantly larger fraction of prevalences close to 1, pointing towards lower mobility within that building. Hence, for SBldg, we find that the distribution follows a power law only for prevalence metrics in the range $[0, 0.9]$. The difference is most probably due to the smaller size of the building: there are only 10 access points in SBldg.

Given these distributions, for each building, we characterize each user with two numbers: the maximum prevalence and the median prevalence. For a given maximum prevalence, the median is inversely proportional to the mobility of the user. The more access points a user visits, the lower the median prevalence.

With these two measures, we categorize users into five groups as shown in Table 2. By increasing mobility, the categories are: stationary, occasionally mobile, regular, somewhat mobile, and highly mobile. Stationary users stay with a single access point almost all the time so their maximum prevalence is high and their median is equal to their maximum. Occasionally mobile users spend most of their time with a single access point and visit others infrequently. Their maximum prevalence is high, while their median prevalence is low. As the user spends less time with a single access point, their maximum prevalence decreases. We categorize these users as either somewhat mobile or highly mobile. Finally, regular users alternate regularly between a few access points so both their median and maximum prevalences are medium. In all three buildings, around 40% of users are only occasionally mobile. Few users, however, are totally stationary (around 10% for LBldg and MBldg). Users appear more stationary in SBldg, probably due to the small number of access points in the building (only 10). A considerable fraction of users (10% to 40%) is somewhat mobile, but only a few users are highly mobile.



(a) Guest persistence



(b) Home persistence

Figure 20: Probability distribution of persistence values up to 400 min for home and for guest users in LBldg. Above 400 min, the probability is close to negligible. Since our sampling interval is 5 min and often stretches slightly above that value, most persistence values up to 5 min are rounded up to the 5 min-10 min interval.

Maximum Prevalence (P_{max})	**Median Prevalence** (P_{med}) **Low** $P_{med} \in [0, 0.25)$	**Med** $P_{med} \in [0.25, 0.50)$	**High** $P_{med} \in [0.50, 1]$
Low $P_{max} \in [0, 0.33)$	highly mobile (4%,6%,0%)	N/A	N/A
Medium $P_{max} \in [0.33, 0.66)$	somewhat mobile (38%,29%,11%)	regular (10%,11%,13%)	N/A
High $P_{max} \in [0.66, 1]$	occ mobile (39%,44%,44%)	N/A	stationary (9%,10%,32%)

Table 2: User categorization based on prevalence metrics. For each building, the fraction of users who belong to each category is indicated under the category name (LBldg, MBldg, and SBldg respectively).

5.3 Persistence

Prevalence has one major limitation: It does not take into account the amount of time users stay associated with access points. A user who spends a week with an access point and another week with another access point will have the same prevalence metrics as a user who continuously moves between two access points. To complement the prevalence metric, we compute user *persistence* at various locations. The persistence is the amount of time that a user stays associated with an access point before moving to another access point or leaving the network. Since we poll access points every 5 to 10 min, we see only visits longer than that interval.

Given our distinction between home and guest locations, we plot the probability distribution of persistence separately for each group (and for each building). Figure 20 shows the probability distribution of persistence values up to 400 min separated between home locations and guest locations. Both distributions follow power laws. For guest users, the exponents are higher indicating that *shorter sessions are more frequent.* Additionally, we note a knee in the probability distribution of guest users, which indicates two different trends in persistence value distributions. The knee appears around 100 min. After that threshold, the power law distribution becomes even steeper indicating that longer sessions become even rarer above that threshold. The two trends that appear may be explained as follows. Short sessions are due to users moving around, attending talks which last between 20 min and one hour, and also users taking breaks in common rest areas. Hence, for up to an hour, various session durations are registered. Longer sessions are mostly due to meetings that last between an hour and two hours but hardly ever take longer than that. We find an almost identical fit for both MBldg and SBldg distributions with a slightly higher exponent after the 100 min knee for MBldg and slightly different constants.

Balachandran et al. [3] find that user session durations follow a General Pareto Distribution with a shape parameter of 0.78. This is equivalent to a power law distribution with exponent 1.78 (i.e., $\frac{1}{x^{1.78}}$). This distribution is closest to what we obtain for guest users, which is what we might expect since during a conference users are not at their normal home locations.

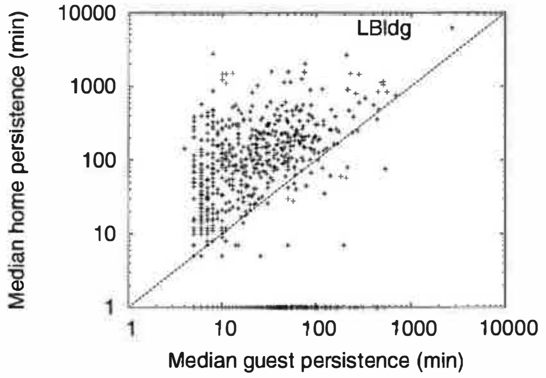


Figure 21: Scatterplot of median guest and median home persistence values for users who visited LBldg. Users without a home location in LBldg are assigned a home median of 1 min so they are also represented on the graph. Home median persistence is almost always greater than guest median persistence, often by as much as one order of magnitude.

Finally, we compare each user's median persistence at home and at guest locations. Figure 21 shows the scatterplot obtained for LBldg. A few users have a home median persistence lower than a guest median persistence. These are mostly users regularly alternating among a few access points. In most cases, persistence at home is equal to, or even one or two orders of magnitude greater than, persistence at guest locations. For users without a home location, median persistence varies within the same range as for other users.

In conclusion, the primary characteristic of user mobility is that many users spend a large fraction of their time in a single location. They visit this location frequently and stay for long periods of time. When they move away, they do not reduce their data transfer rates, but they spend short periods of time at any location and they do not visit the same location frequently.

6 Discussion

In this section, we discuss possible applications of user mobility and network usage characteristics to wireless network deployment, to workload generation, and to application design.

6.1 Wireless network design and deployment

Several approaches recently introduced new algorithms to relieve "hot-spots" and dynamically balance load among access points. Cisco access points [7] balance load between each other (within an overlapping cell) using the number of users, their error rates, and signal strength. Balachandran et al. [2] improve load balancing by explicitly re-directing users to satisfy pre-negotiated bandwidth range service agreements. Balancing users across access points is important. As shown in Section 4.2.5, even in a well provisioned network, access points often experience periods of high demand lasting a few minutes. In our trace, most of these peaks affected a single access point at a time. Therefore, quickly load balancing users could often unload one or two heavily-loaded access points.

Our analysis provides further information that may be helpful in designing load balancing algorithms: a) we find high personal differences in network usage; b) users with high average transfer rates represent a small fraction of all users; and c) any access point sees only a small fraction of all users. Additionally, location seems to play some role in user's level of activity. Given these network usage characteristics, we propose that access points keep mobility and network usage figures for each moderately active client. They could then better react to overload because they would know how long associated users were likely to stay and what amount of resources they were likely to require. Of course, to protect user privacy, access points should neither make this data openly available nor communicate it to other access points.

For each associated user, access points already keep counters of bytes and packets transferred. For users transferring more than 2Mbps or 3Mbps, each access point could preserve a running average of data transferred as well as the user's peak transfer rates. Each access point could also preserve a running average of persistence values, updating them as follows every time a user de-associates from an access point: $P_{user,t} = \alpha * P_{user,t-1} + (1 - \alpha)D_{user}$, where $P_{user,t-1}$ is the previous average persistence for a user, D_{user} is the amount of time the user remained associated this time, and $\alpha \in [0, 1]$ is a factor adjusting the importance of history over the latest value. The average transfer rate, $X_{user,t}$, could be computed in a similar manner. In a situation of overload, an access point could choose to re-direct a user to another location if this user had a history of using a lot of resources (high X), but not staying very long (low P). The access point would not waste resources redirecting users that are always idle. It would also avoid re-directing users who stay with the access point for prolonged periods of time (perhaps those who have their office there).

Access point popularity is another useful metric for network deployment. To provide coverage, we find that some access points are deployed in locations where they are seldom used. These access points could self-tune to reduce their power consumption or increase their coverage (while decreasing maximum data rates). Additionally, access point could compute and compare user persistence and prevalence metrics to determine their relative popularity. Such relative popularity metrics, coupled with load metrics, would help system administrators determine the most appropriate locations for new access points. Ideally, system administrators want to deploy extra access points before users see performance degradation, so they need other metrics to determine where new access points should be added.

6.2 Refining workload generation

Several tools exist to simulate wireless networks [11, 12, 14]. They model characteristics of the wireless network quite accurately, but they require users to dynamically define the location of a node or to define trajectories. However, tools could use the concept of "home locations" as well as power law distributions for persistence and prevalence to simulate user mobility automatically. Persistence could serve to determine how long a user stays at a location on average, whereas prevalence could serve to determine which location the user visits.

Additionally, we find it reasonable to use scaled down population models to determine the number of users present on the simulated network at any point in time.

6.3 Guiding application design

Knowledge of network usage characteristics may also prove helpful in designing applications for mobile environments. For example, since users spend a large fraction of time at their home location the design of mobile systems might benefit from optimizations for this particular usage pattern. An application may, for example, keep information about each user at their home location.

Additionally, in all environments studied, most users visit only a few locations during a day, and spend a large fraction of their time at their home location before visiting other locations on subsequent days. This specific mobility pattern should also influence design decisions. For example, when synchronizing application data, it would be an appropriate design choice to designate the home location as the "master" replica.

7 Conclusion

In this paper, we presented the analysis of results from a four week trace gathered in a large corporate environment and showing the network usage of 1366 different users. Analyzing mobility and network usage, we find several characteristics, many of which are shared by users in other environments such as university campuses and public networks.

In spite of increasing popularity of wireless networks, the number of days each user appears on the network is highly variable among different users. However, the general patterns in the numbers of users per day and the number of users per hour follow a regular office schedule.

Load is unevenly distributed across access points. Some, located in popular areas such as large auditoriums, often see high numbers of users simultaneously associated with them (up to 30). Others, located in less visited areas, are usually idle. We find that the amount of traffic at an access point is weakly dependent on the number of users present or the time of day. Most traffic is due to a small fraction of active users: the most-active 10% of users are responsible for more than 40% of the data transferred at 30% of locations. Load is also somewhat related to access point location, as we find that location impacts user transfer rate significantly for 30% of active users. Additionally, user's average level of activity follows a power law distribution.

We introduce *persistence* and *prevalence* to characterize and classify user mobility. Probability distributions of both metrics follow power law distributions. Persistence at guest locations also has a higher exponent than persistence at home locations, clearly showing that users associate with access points longer when staying at their home locations. Using prevalence, users can be categorized into mostly stationary, occasionally mobile, regular, somewhat mobile, and highly mobile. We find that 50% to 80% of users fall into the occasionally and somewhat mobile categories. Finally, we find that mobility does not influence user level of activity on the network. However, most devices in our study were laptops; mobility results may become different as PDAs and other small devices become more popular.

We plan to repeat this study, using SNMP in combination with syslog and tcpdump as well as monitoring software on mobile devices. Our goal is to get more detailed information on network usage and develop more detailed models of both mobility and network usage.

Acknowledgmenents

We thank Apratim Purakayastha for his helpful suggestions throughout the project. We thank Drew Wyskida for helping us gain access to the access points. We gratefully acknowledge the help of David Kotz with SNMP and data collection scripts. We thank Hari Balakrishnan and Chuck Blake for their suggestions on result presentation and analysis. We also thank everyone who read and commented on draft versions of this paper.

References

[1] B. Alexander and S. Snow. Preparing for wireless LANs: Secrets to successful wireless deployment. Packet Magazine. Cisco Systems. `http://www.cisco.com/warp/public/784/packet/apr02/p36-cover.html`, April 2002.

[2] Anand Balachandran, Paramvir Bahl, and Geoffrey M. Voelker. Hot-spot congestion relief and user service guarantees in public-area wireless networks. In *Proc. of the 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2002)*. IEEE Computer Society, June 2002.

[3] Anand Balachandran, Geoffrey M. Voelker, Paramvir Bahl, and P. Venkat Rangan. Characterizing user behavior and network performance in a public wireless LAN. In *Proc. of ACM SIGMETRICS'02*. ACM Press, June 2002.

[4] J. Case, M. Fedor, M. Schoffstall, and J. Davin. RFC1157: A simple network management protocol (SNMP). `http://ietf.org/rfc/rfc1157.txt?number=1157`, 1990.

[5] Alex Hills. Wireless Andrew. *IEEE Spectrum*, 36(6):49–53, June 1999.

[6] William W. Hines and Douglas C. Montgomery. *Probability and Statistics in Engineering and Management Science. Third Edition.* John Wiley and Sons, 1990.

[7] "Cisco Systems Inc.". Data sheet for Cisco Aironet 350 Series access points. `http://www.cisco.com/warp/public/cc/pd/witc/ao350ap/prodlit/`, July 2001.

[8] David Kotz and Kobby Essien. Analysis of a campus-wide wireless network. In *Proc. of the Eigth Annual Int. Conf. on Mobile Computing and Networking (MobiCom)*. ACM Press, September 2002.

[9] David Kotz and Kobby Essien. Characterizing usage of a campus-wide wireless network. Technical Report TR2002-423, Dept. of Computer Science, Dartmouth College, March 2002.

[10] Kevin Lai, Mema Roussopoulos, Diane Tang, Xinhua Zhao, and Mary Baker. Experiences with a mobile testbed. In *Proc. of the Second International Conference on Worldwide Computing and its Applications (WWCA'98)*, March 1998.

[11] ns-2. The Network Simulator. `http://www.isi.edu/nsnam/ns/`.

[12] Inc. OPNET Technologies. OPNET Modeler. `http://www.opnet.com/products/modeler/home.html`, 2002.

[13] Vern Paxson. End-to-end routing behavior in the internet. In *Proc. of the ACM SIGCOMM Conference*. ACM Press, August 1996.

[14] Rice Monarch Project. `http://www.monarch.cs.rice.edu/cmu-ns.html`, 1999.

[15] Diane Tang and Mary Baker. Analysis of a local-area wireless network. In *Proc. of the Sixth Annual Int. Conf. on Mobile Computing and Networking (MobiCom)*. ACM Press, August 2000.

[16] Diane Tang and Mary Baker. Analysis of a metropolitan-area wireless network. *Wireless Networks*, 8(2/3):107–120, 2002.

[17] ZDNet News. 802.11 group looking for more 'hot spots'. `http://zdnet.com.com/2100-1105-936659.html`, June 2002.

GPRSWeb: Optimizing the Web for GPRS Links

Rajiv Chakravorty, Andrew Clark and Ian Pratt
E-mail: {firstname.lastname}@cl.cam.ac.uk
University of Cambridge Computer Laboratory,
JJ Thomson Avenue, Cambridge CB3 0FD, U.K.

ABSTRACT

The General Packet Radio Service (GPRS) is being deployed by GSM network operators world-wide, and promises to offer users "always-on" data access at bandwidths comparable to that of conventional fixed-line telephone modems. Unfortunately, many users have found the reality to be rather different, experiencing very disappointing performance when, for example, browsing the web over GPRS.

In this paper we investigate what causes the HTTP protocol and its underlying transport TCP to underperform in a GPRS environment. We examine why certain GPRS network characteristics interact badly with TCP to yield problems such as: link under-utilization for short-lived flows, excess queueing for long-lived flows, ACK compression, poor loss recovery, and gross unfairness between competing flows. We also show that many web browsers tend to be overly aggressive, and by opening too many simultaneous TCP connections can aggravate matters.

We present the design and implementation of GPRSWeb – a mobile HTTP proxy system that mitigates many of the performance problems with a simple software update to a GPRS mobile device. The update is a 'client proxy' that sits in the mobile device, and communicates with a 'server proxy' located at the other end of the GPRS link close to the wired-wireless border. The dual proxy architecture collectively implements a number of key enhancements – an aggressive caching scheme that employs content-based hash keying to improve hit rates for dynamic content, a preemptive push of web page support resources to mobile clients, resource adaptation to suit client capabilities, delta encoded data transfers, DNS lookup migration, and a UDP-based reliable transport protocol that is specifically optimized for use over GPRS. We show that these enhancements result in significant improvement in overall WWW performance over GPRS.

1. INTRODUCTION

World over, GSM cellular networks are being upgraded to support the General Packet Radio Service (GPRS). GPRS offers an "always on" connectivity to mobile users, with wide-area coverage and data rates comparable to that of conventional fixed-line telephone modems. This holds the promise of making ubiquitous mobile access to IP-based applications and services a reality.

However, despite the momentum behind GPRS, surprisingly little has been done to evaluate WWW performance over GPRS. There are some interesting simulation studies [1, 4] on TCP performance, but we have found actual deployed network performance to be somewhat different.

Some of the web performance issues observed with GPRS are shared, to some extent, with wireless LANs like 802.11b (*WiFi*), satellite systems, and other wide-area wireless schemes such as Metricom Ricochet and Cellular Digital Packet Data (CDPD). However, we feel that GPRS presents a particularly challenging environment for achieving good application (web) performance.

Past research has investigated TCP (and also HTTP) performance over a number of wide-area wireless links such as Ardis, Metricom Richochet, CDPD and GSM. However, the real inhibitors to a better web browsing experience are typically related to the underlying network characteristics, which as we shall see, are somewhat different for GPRS.

In this paper we set out to explore questions like:

1. What are the "typical" GPRS network characteristics?
2. What are the practical performance problems using TCP and HTTP over GPRS?
3. What overall benefit can we achieve using various application level optimization schemes over GPRS?

In this paper, we present our practical experiences over production GPRS networks, and our attempts to build a system that optimizes WWW performance over GPRS. After a brief overview on GPRS in the next section, we summarize our work to characterize GPRS link behaviour in section 3. Section 4 identifies particular problems experienced by TCP flows over GPRS, and section 5 examines how these are exacerbated by application-layer protocols such as HTTP.

In section 6, we present the design and implementation of our GPRSWeb proxy system – a dual-proxy system architecture consisting of a 'client proxy' and a 'server proxy'. While the GPRSWeb client proxy resides in the mobile device, the GPRSWeb server proxy is located in the network in close vicinity to the wired-wireless border. GPRSWeb aims to improve WWW performance with an optimized transport protocol specifically tailored for GPRS, an advanced caching scheme, server controlled *parse-and-push* functionality, data compression, and document delta encoding. GPRSWeb requires no instrumentation or modifications to be made to either web browsers or servers.

Section 7 discusses our experimental test bed setup and in section 8 we present an evaluation of the performance our proxy system. We conclude with a discussion of related work and summarize our experience with the system.

2. THE GPRS OVERVIEW

GPRS is a bearer service for GSM - a wireless extension to packet data networks. Two new nodes (see, figure 6) have been added to the traditional GSM network to support GPRS: the SGSN (Serving GPRS Support Node) and GGSN (Gateway GPRS Suport Node). The SGSN node acts as a packet switch that performs signaling similar to a mobile switching center (MSC) in GSM, along with cell selection, routing and handovers between different Base Switching Centers (BSCs). It also controls the mobile terminal's access to the GPRS network and routes packets to the appropriate BSC. The GGSN is the gateway between the mobile packet routing of GPRS and the fixed IP routing of the Internet.

A mobile terminal (MT) wishing to use GPRS will first *attach* itself to the network through a signaling procedure. The *attach* procedure can be performed either when the MT is switched on or when the user wishes to transfer packet data. Depending upon the type of GPRS device class, it can connect either to circuit switched or to packet switched services, or both simultaneously [1]. Mobile terminals are classified according to the number of time slots they are capable of operating on simultaneously. For example, many current GPRS devices are classified as "3+1" meaning that at any given time they can listen to 3 downlink channels (from base station to mobile), but can only transmit on 1 uplink channel to the base station.

GPRS copes with a wide range of radio conditions by making use of 4 different coding schemes (CS-1 TO CS-4) [1][5] with varying levels of FEC (forward error correction). Most of the currently deployed GPRS networks support only CS-1 and CS-2 [34] – the other two are not used as error rates would be typically too high to be useful. CS-4 removes FEC correcting capabilities altogether. The effective GPRS data rate is slightly less, due to protocol header overhead and signalling messages. The RLC (radio link control) layer attempts to provide reliable in-order delivery of packets.

Radio resources (TDMA time slots) of a cell are shared between all GPRS and GSM mobile stations located within a cell. Most network operators typically configure the network to give GSM (voice) calls strict priority over GPRS for time slot allocation. The time slots available for GPRS use, known as packet data channels (PDCHs), are then dynamically allocated (using the *capacity on demand* principle [5]) between mobile terminals with data to send or receive. GPRS can multiplex time slots between different users, and can also allow multiple time slots to be used in parallel to increase bandwidth to/from a particular mobile terminal.

The latest GPRS Release (1999) defines several QoS parameters to meet the application requirements for different levels of network QoS. However, currently deployed networks typically only support a single *best-effort* service class [34]. Further information about GRPS network design and operation can be found in [1-5].

3. GPRS LINK CHARACTERIZATION

GPRS [1, 5], like other wide-area wireless networks, exhibits many of the following characteristics: low and fluctuating bandwidth, high and variable latency, and occasional link 'blackouts' [3, 7]. To gain clear insight into the characteristics of the GPRS link, we have conducted a series of link characterization experiments. The-se have been repeated under a wide range of conditions, using different models and manufacturer of handsets, and different network operators located in several European countries. We have found no major performance differences between the network operators, and variation between different handsets of similar GPRS device class is minimal. Details of how these tests were conducted (uplink and downlink latency measurements, tools etc.) can be found in [2]. [33] also gives a comprehensive description on GPRS link characterization in the form of a separate technical report. Below, we enunciate some key findings:

High and Variable Latency:- GPRS link latency is very high, 600ms-3000ms for the downlink and 400ms-1300ms on the uplink. Round-trip latencies are 1000ms or more. The delay distribution is shown in figure 1. The link also has a strong tendency to 'bunch' packets; the first packet in a burst is likely to be delayed and experience more jitter than following packets. This indicates that a substantial proportion of the latency is incurred when the link to a mobile terminal transitions from previously being idle [2]. Packets that are already queued for transmission can then follow the first out over the radio link without incurring additional jitter. The additional latency for the first packet is also typically due to allocation time of the temporary block flows (TBFs). Since most current GPRS terminals allocate and release TBFs immediately (implementation based on GPRS 1997 release), applications (such as TCP) that can transfer temporally-separated data (and ack) packets may end up creating many small TBFs that can each add some delay (approx. 100-200msec) during data transfer.

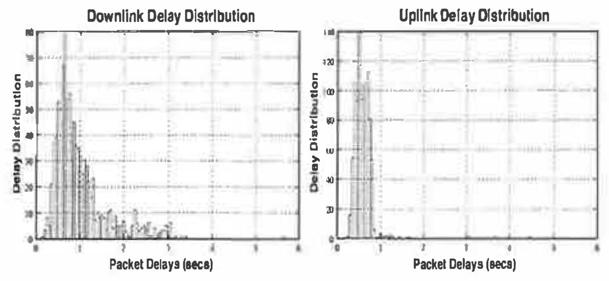


Figure 1: Single packet time-in-flight delay distribution plots showing (a) downlink delay (b) uplink delay distribution. Measurements involved transfer of 1000 packets with random intervals > 4s between successive packet transfers.

Fluctuating Bandwidth:- We observe that signal qu-ality leads to significant (often sudden) variations in perceivable bandwidth by the receiver. Sudden signal quality fluctuations (good or bad) commensurately impacts GPRS link performance. Using a "3+1" GPRS phone such as the Ericsson T39 (3 downlink channels, 1 uplink), we observed a maximum raw downlink throughput of about 4.15 KB/s (33.2 Kb/s), and an uplink throughput of 1.4 KB/s (11.2 Kb/s). Using a "4+1" phone, the Motorola T280, we measured an improved maximum bandwidth of 5.5 KB/s (44 Kb/s) in the downlink direction. If CS-2 coding scheme was used, then using a '3+1' and "4+1" phone, we could achieve a theoritical maximum of 40.2 Kb/s and 53.6 Kb/s respectively. Some other factors contribute to throughput values lower than the maximum possible, see [2, 4].

Packet Loss:- The radio link control (RLC) layer in GPRS uses an automatic repeat request (ARQ) scheme that works aggressively to recover from link layer losses. Thus, higher-level protocols (like IP) rarely experience non-congestive losses. Packets can be lost over the GPRS link during, (1) deep fading leading to bursty losses, and (2) cell re-selections due to the cell (or routing area) update procedure resulting in link a 'black-out' condition. In both cases, consecutive packets in a window are usually lost.

Link Outages:- Link outages are common while moving at speed, or when passing through tunnels or other radio obstructions. Nevertheless, we have also noticed outages during stationary conditions. The observed outage interval will typically vary between 5 and 40s. Sudden signal quality degradation, prolonged fades and intra-zone handovers can lead to such link blackouts. When link outages are of short duration, packets are simply delayed and are lost in few cases. In contrast, when outages are of higher duration there tend to be burst losses. We have also observed specific cases of link resets, where a mobile terminal would stall and stop listening to its temporary block flow (TBF). In such cases, we had to terminate and restart the point-to-point (PPP) session.

4. TCP PERFORMANCE OVER GPRS

In this section we discuss TCP performance problems over GPRS. In particular, we concentrate on connections where the majority of data is being shipped in the downlink direction, as this corresponds to the prevalent behaviour of most mobile applications, such as web browsing, file download, reading email, news etc. A more comprehensive description on TCP performance problems over GPRS can be found in [3].

TCP Start-up Performance:- Figure 2(a) shows a close up of the first few seconds of a connection, displayed alongside another connection under slightly worse radio conditions. An estimate of the link bandwidth delay product (BDP) is also marked, approximately 10KB[1]. For a TCP connection to fully utilize the available link bandwidth, its congestion window must be equal or exceed the BDP of the link. We can observe that in the case of good radio conditions, it takes over 7 seconds to ramp the congestion window up to a value of link BDP from when the initial connection request (TCP's SYN) was made. Hence, for transfers shorter than about 18KB, TCP fails to exploit the meagre bandwidth that GPRS makes available to it. Since many HTTP objects are smaller than this size, the effect on web browsing performance can be dire.

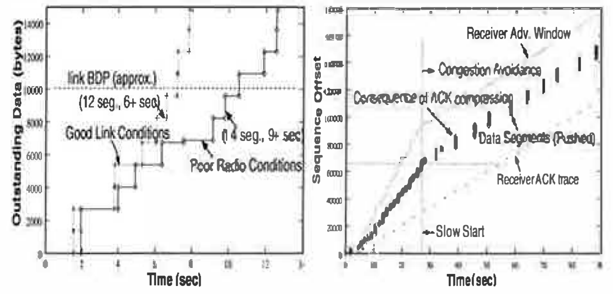


Figure 2: Plot (a) shows that slow-start takes over 7 seconds to expand the congestion window sufficiently to enable the connection to utilise the full link bandwidth. (b) shows the characteristic exponential congestion window growth due to slow-start (SS).

ACK Compression:- A further point to note in figure 2(b) is that the sender releases packets in bursts in response to groups of four ACKs arriving in quick succession. Receiver-side traces show that the ACKs are generated in a smooth fashion in response to arriving packets. The 'bunching' on the uplink is due to the GPRS link layer. This effect is not uncommon, and appears to be an unfortunate interaction that can occur when the mobile terminal has data to send and receive concurrently. ACK bunching or compression not only skews upwards the TCP's RTO measurement but also affects its self-clocking strategy. Sender side packet bursts can further impair RTT measurements.

Excess Queuing:- Due to its low bandwidth, the GPRS link is almost always the bottleneck of any TCP connection, hence packets destined for the downlink get queued at the gateway onto the wireless network (known as the GGSN node in GPRS terminology, see figure 9). However, we found that the existing GPRS infrastructure offers substantial buffering: UDP burst tests indicate that over 120KB of buffering is available in the downlink direction. For long-lived sessions, TCP's congestion control algorithm could fill the entire router buffer before incurring packet loss and reducing its window. Typically, however, the window is not allowed to become quite so excessive due to the receiver's flow control window, which in most TCP implementations is limited to 64KB unless *window scaling* is explicitly enabled. Even so, this still amounts to several times the BDP of unnecessary buffering, leading to grossly inflated RTTs due to queueing delay. Figure 3 (b) shows a TCP connection in such a state, where there is 40KB of outstanding data leading to a measured RTT of tens of seconds. Excess queueing exacerbates other issues:

[1]The estimate is approximately correct under both good and bad radio conditions, as although the link bandwidth drops under poor conditions the RTT tends to rise.

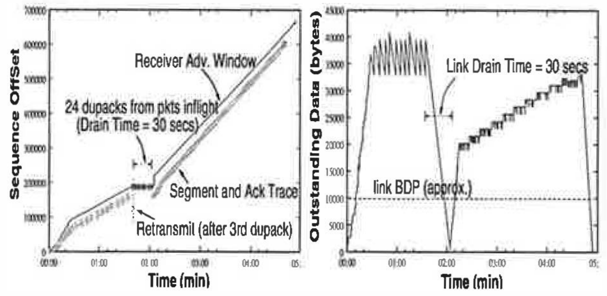


Figure 3: Case of timeout due to a dupack(sack). Plot (a) shows the sender sequence trace and (b) shows corresponding outstanding data.

- **Inflated Retransmit Timer Value.** RTT inflation results in an inflated retransmit timer value that impacts TCP performance, for instance, in cases of multiple loss of the same packet [6].

- **SYN timeout.** Excess queuing caused by long-lived flows often results in attempts to establish new connections timing-out before completing the 3-way handshake. [6].

- **Problems of Leftover (Stale) Data.** For downlink channels, the queued data may become obsolete when a user aborts a web download and abnormally terminates the connection. Draining leftover data from such a link may take many seconds.

- **Higher Recovery Time.** Recovery from timeouts due to dupacks (or sacks) or coarse timeouts in TCP over a saturated GPRS link takes many seconds. This is depicted in figure 3(a) where the *drain time* is about 30s.

TCP loss recovery over GPRS:- Figure 3(a)-(b) depicts TCP's performance during recovery due to reception of a dupack (in this case a SACK). The point to note here is the very long time it takes TCP to recover from the loss, on account of the excess quantity of outstanding data. Fortunately, use of SACKs ensures that packets transferred during the recovery period are not discarded, and the effect on throughput is minimal. This emphasises the importance of SACKs in the GPRS environment. In this particular instance, the link condition happened to improve significantly just after the packet loss, resulting in higher available bandwidth during the recovery phase.

Fairness between flows:- Excess queueing can lead to gross unfairness between competing flows. Figure 4 shows a file transfer (f2) initiated 10 seconds after transfer (f1). When TCP transfer (f2) is initiated, it struggles to get going. In fact it times out twice on initial connection setup (SYN) before

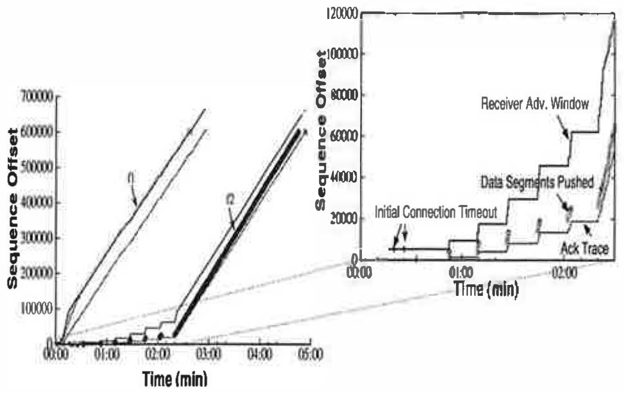


Figure 4: Close-up of time sequence plots for two concurrent file transfers over GPRS, where f2 was initiated 10 seconds after f1.

being able to send data. Even after establishing the connection, the few initial data packets of f2 are queued at the CGSN node behind a large number of f1 packets. As a result, packets of f2 perceive very high RTTs (16-20 seconds) and bear the full brunt of excess queueing delays due to f1. Flow f2 continues to badly underperform until f1 terminates. Flow fairness turns out to be an important issue for web browsing performance, since most browsers open multiple concurrent HTTP connections [10]. The implicit favouring of long-lived flows often has the effect of delaying the "important" objects that the browser needs to be able to start displaying the partially downloaded page, leading to decreased user perception of performance.

5. WWW PERFORMANCE OVER GPRS

The results from the preceding section show that TCP performs poorly in a wide-area wireless GPRS environment. In this section, we briefly review the key issues related to browser, and specifically, web performance over GPRS. Further information related to web performance issues over GPRS can be found in [7].

Figure 5 shows a sample connection setup overhead for a web connection. Typically, the setup overhead in a web connection could entail two round-trips. In the first round-trip, a web client resolves the requested Uniform Resource Identifier (URI) with a check to a local Domain Name Server (DNS) cache for an entry to the requested URL. If it is present in the local DNS cache and has not yet timed-out, the cache entry is directly used to prevent an unnecessary DNS server lookup. However, if the local cache search fails, the client then makes a request to the DNS server, which in the GPRS case, is located on the other end of the wireless link. As RTTs over GPRS are high, a significant DNS lookup overhead (higher than a single GPRS RTT) is entailed in the process.

In the second round-trip, however, the web client in a mobile device initiates a TCP connection with the remote server. As again in this case, every TCP connection will have to proceed through a 3-way TCP handshake, which means an ad-

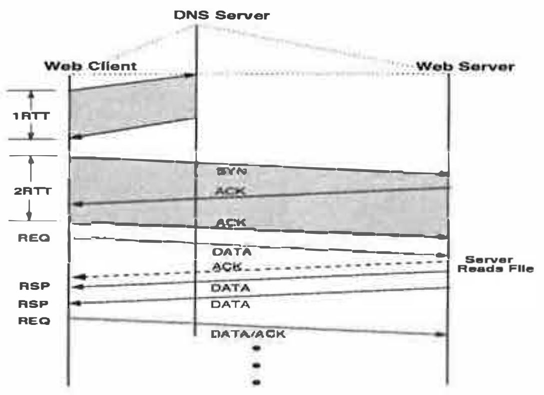


Figure 5: Web Connection SetUp Overhead

ditional RTT is incurred before the connection is used. Once this is done, the client can proceed making a request to the server followed by normal data transfer from the server. Thus the total delay for the first TCP connection to successfully initiate the first HTTP request can be as high as 2 GPRS round trips.

Browser behaviour is obviously crucial to a web performance over any given network. Unfortunately, most current web browsers are tuned for LAN environments and often perform poorly in a resource restricted setting. In [7], experiments conducted show that web browsers (e.g. mozilla) tend to open multiple concurrent TCP connections over a link simultaneously. The inherent nature of TCP's congestion control algorithm implies that N connections will be N times more aggressive when compared to a single TCP connection when sharing a bottleneck link. Typically, web browsers that open a number of concurrent TCP connections do so to grab a greater share of the link bandwidth. Also, with more connections, browsers implicitly avoid *head-of-line* (HOL) [14] blocking problems. An aggressive browser will obviously reap benefits over conventional high bandwidth links shared by many users.

Using multiple connections over 'long-thin' GPRS links can be deleterious: First, protocol control (SYNs/ACKs-/FINs) overhead associated with higher numbers of connections is high. This is further exacerbated by the overhead of the protocol headers (i.e. TCP+IP+SNDCP+L-LC=55 bytes, as in [34]) even for data packets that are exchanged over the link. Secondly, the 3-way handshake delay while establishing a TCP connection can be significant due to the high latency of GPRS links. Further, it can take only a few RTTs for multiple concurrent connections to exceed the GPRS CGSN router downlink bandwidth-delay product (BDP) value. The exponential nature of the slow-start phase combined with packets from multiple flows can quickly lead to excess queuing over the downlink. As a result, any subsequent new TCP connection will have a high chance of timing out during its initial connection request phase. New connections will endure high RTTs, causing them to severely underperform, with an additional probability of spurious timeouts. Experiments from [7] over production GPRS networks show that aggressive web browsers tend to saturate the downlink GPRS GGSN buffers. This has been shown to negatively impact web performance over GPRS.

Many of the widely deployed web browsers continue to use non-persistent connections (HTTP/1.0), employing a new TCP connection for every object downloaded. Use of HTTP/-1.1 persistent, connections, where the browser and server employ the same TCP connection for transfer of multiple objects are gradually becoming more widespr-ead, and help reduce connection setup delay and overhead. However, the full benefits of HTTP/1.1 can not be realised without making use of *pipelining*, where multiple outstanding requests are permitted on the same connection. Without pipelining, a RTT delay is incurred between objects on the same connection, and worse, slow-start must be performed at the start of every object since the connection will have gone idle. Unfortunately, use of pipelining is currently almost non existent. Experiments in [7] show that HTTP request pipelining can improve web performance over GPRS.

6. THE GPRSWEB PROXY MODEL

Having identified the causes of poor WWW performance in a GPRS environment, we now report on our attempts to overcome them. Clearly, performance could be improved by making modifications to the HTTP and TCP protocols to better suit the GPRS environment. However, any approach that relies on modifications to web servers or web browsers would at best take years to achieve wide-spread deployment. Our approach has been to use the existing HTTP proxy mechanism to enable us to insert a pair of translating proxy servers in to the HTTP request/response stream that together implement a number of techniques to enhance performance. This enables GPRSWeb to be both browser and server independent.

GPRSWeb uses a pair of special proxy servers, located on either side of the GPRS link. Between the proxies, a custom protocol is employed to reduce traffic volume over the GPRS link and attempt to mitigate the effects of GPRS's high RTT. The 'client proxy' must be installed on the mobile device. As part of the installation, the web browser is configured to route all HTTP requests through this proxy via a local TCP connection using the traditional *loopback* interface. As shown in figure 6, the client proxy communicates with a 'server proxy', located on the other end of the GPRS link. The server proxy makes requests to the wired network on behalf of the client, and sends back responses. The server proxy is capable of servicing large numbers of simultaneous mobile clients. The cache content at the server proxy is shared.

The GPRSWeb proxy model implements the following mechanisms to improve performance:

GPRSWeb Protocol. Due to the problems identified earlier, we do not use TCP as the transport protocol between the proxies either side of the GPRS link. Instead, we use a custom transport protocol (which we call GPRSWeb protocol hereafter) that runs over UDP and implements ordered, reliable, message transfer. The protocol is optimised for GPRS link characteristics, and minimises link traversals and responds efficiently in the event of the patterns of packet loss

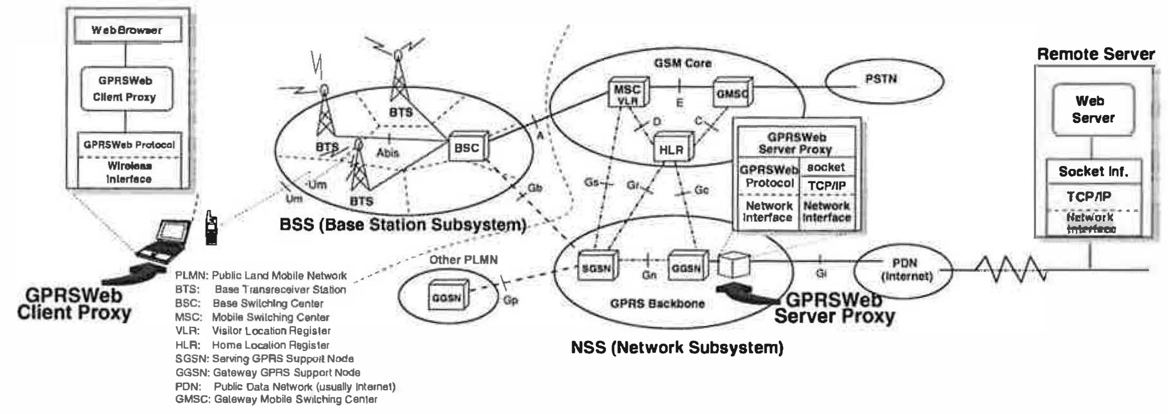

Web Browser
GPRSWeb Client Proxy
GPRSWeb Protocol
Wireless Interface
GPRSWeb Client Proxy
BTS
BSC
Abis
Um
BSS (Base Station Subsystem)
GSM Core
MSC VLR
GMSC
HLR
PSTN
GPRSWeb Server Proxy
GPRSWeb Protocol
socket
TCP/IP
Network Interface
SGSN
GGSN
GPRS Backbone
Other PLMN
PDN (Internet)
GPRSWeb Server Proxy
NSS (Network Subsystem)
Remote Server
Web Server
Socket Inf.
TCP/IP
Network Interface
PLMN: Public Land Mobile Network
BTS: Base Transreceiver Station
BSC: Base Switching Center
MSC: Mobile Switching Center
VLR: Visitor Location Register
HLR: Home Location Register
SGSN: Serving GPRS Support Node
GGSN: Gateway GPRS Support Node
PDN: Public Data Network (usually Internet)
GMSC: Gateway Mobile Switching Center

Figure 6: GPRSWeb System Architecture and Components

we commonly observe. It achieves substantially better link utilization than TCP.

Extended Caching. Client-side caching improves performance by eliminating some network round-trips and reducing the amount of data exchanged over the GPRS link. However, traditional browser caches do not yield the full potential benefit due to the nature of the HTTP caching mechanism, and pessimistic cache control directives contained in many web pages.

The GPRSWeb client proxy implements a client-side cache that replaces the browser's persistent (disk) cache. A custom caching protocol is used between the client and server proxies that enables better hit rates by using `SHA-1` fingerprints [29] of objects to determine whether they have actually changed or not. The protocol thus eliminates unnecessary object transfers over the GPRS link, and makes better use of the limited size cache available in the mobile device. The server-side proxy also implements a traditional HTTP cache to reduce bandwidth requirements on the wired network, and thus can take the place of existing proxy caches that are already commonly deployed by ISPs.

Data Compression and Delta Encoding. GPRSWeb also compresss data before sending it over the wireless link, reducing transfer size and thereby improving response time. Data is compressed using *gzip* compression, unless it is already in a compressed format (e.g. JPEG images, Zip Archives). A separate string table is used for HTTP headers, resulting in better compression. When the server-side proxy detects that a previously cached object has been updated, it tries using the VCDiff [31] algorithm to encode the differences between the old and new objects. The compressed *deltas* [15] are sent in place of the new version if they would result in a smaller transfer.

Parse-and-Push Operation. Most web pages contain a number of images and other objects that make up the page structure, e.g. button graphics, spacers, style sheets, frames etc. These objects are requested by a browser after parsing the HTML documents. A round trip delay is normally incurred before transfer of these objects can commence. The *parse-and-push* mechanism in the GPRSWeb server proxy parses HTML objects, and pro-actively starts *pushing* objects towards the GPRSWeb client cache if the link would otherwise be idle.

6.1 GPRSWeb Proxy Design

We have designed and implemented the GPRSWeb proxy architecture by splitting the client and server functionality across a number of components, some of which are shared. Figure 7 show the components used in the client and the server structure. We now discuss the components that constitute the GPRSWeb system design.

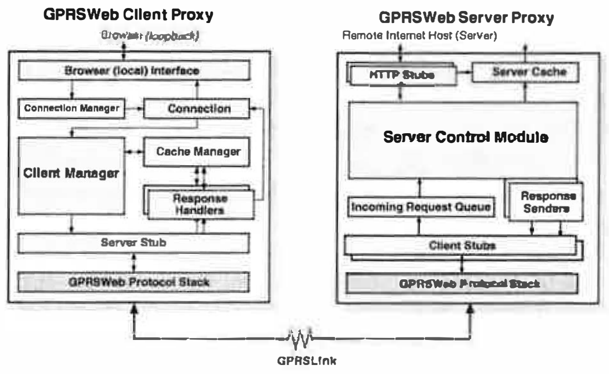


Figure 7: Client and Server Proxy Structure

As shown in the client structure of figure 7, the **Connection Manager** accepts TCP connections from the web browser and passes them to a **Connection** object. This queries the **Client Cache Manager**, and in case of a miss, invokes a **Server Stub** to issue a request to the server proxy. Unique identifiers are assigned to each request made to the server proxy, and are used to invoke a **Response Handler** to process the reply. The response handler also interacts with the cache manager to update cache state as necessary. The object is then returned to the pending browser connection.

In the server proxy, **Client stubs** examine messages received by the protocol stack from a client and takes action depending upon message type. Object request messages are processed by **Server Manager**, which functions very similar to the client manager, but seeks responses from the **Server Cache Manager** and the **HTTP Stubs** if necessary. The HTTP Stubs contact web servers to download or check the freshness of objects. Thus, any DNS lookups required are performed on the server proxy and not over the GPRS link. The client stubs co-ordinate data compression and other optimizations before the response is finally sent back to the client proxy.

6.2 GPRSWeb Protocol

The GPRSWeb transport protocol avoids TCP's connection setup and slow-start costs, and exploits knowledge of GPRS's particular link characteristics to optimise performance.

The basic unit of transfer is the *segment*, each of which is carried in a separate UDP packet. UDP is used to take advantage of the port multiplexing facilities and the UDP checksum. Segments are sequentially numbered, and are of two types: *Normal-priority* and *Low-priority*. Low-priority is typically used for background transfers, e.g. pushing cache updates to a client; normal-priority is used for everything else. The queues are serviced with strict priority. Segments in the low priority queue may be promoted if, for example, the server proxy explicitly receives a request for an object that is currently preemptively being pushed to the client.

Since we expect missing segments to be rare over GPRS (due to the underlying reliable RLC layer) an error recovery strategy based on selective repeat with Negative Acknowledgement (NACKs) scheme is used. In a NACK based scheme, the receiver explicitly indicates to the sender which segments went missing. The NACK based scheme eliminates the retransmission ambiguity problem, and also results in minimal control traffic overhead in the normal case.

Where possible, NACKs are piggybacked on to outgoing segments. If there is no outgoing traffic, an empty 'dummy' segment is created to carry the NACK. As a result of piggybacking, should a NACK be lost, the loss of the carrier segment will be noted, and the NACK re-transmitted with its carrier segment.

The link condition is verified periodically by setting the ACK-able header flag in an outgoing segment (creating a dummy segment if none already exists). The receiving host generates an explicit ACK response, in the same manner it would a NACK. ACK-able messages are generated every few seconds, thus the hosts can detect whether a serious link stall is being experienced. If the client receives no replies for 30 seconds, it attempts to disconnect ad re-attach to the GPRS network; experience shows that this action often brings the link straight back to life.

GPRSWeb uses a connection start-up method very similar to the TCP Accelerated Open (TAO) scheme developed for T/TCP [17], avoiding SYNs/ACK control packets. Each host remembers the segment number last received, and expects to receive the segment following. No handshake is needed, since numbering is assumed to continue from where it left off. Wherever it is necessary to start a new sequence (e.g. a host reboots), initial segments of the new sequence are tagged with the *deltas* between their sequence number, and the base of the new sequence. A host can therefore determine the new sequence base, and issue NACKs for missing segments, even if those missing started a new sequence.

Whereas TCP has to operate over links with widely varying qualities, GPRSWeb can make many more assumptions about the underlying network. Since the GPRS network already implements a mechanism for sharing bandwidth between users, there is no need for the GPRSWeb protocol to implement its own congestion avoidance mechanism. Instead, a simple credit-based flow control scheme suffices.

GPRSWeb initially gives each host credit equivalent to an estimated value of the bandwidth-delay product (BDP) of the link: no slow-start phase is employed. For 3+1 class GPRS devices this initial estimate is 10KB.

This level of outstanding credit is refined over time based on the measured RTT and throughput, typically from timing ACKable segments. The credit value used is set to be 10% higher than the measured RTT-throughput product, to allow the link to remain fully utilised in the presence of typical levels of jitter. Since the outstanding credit is capped in this manner, we avoid the excess queueing long-lived TCP flows cause, and ensure that the buffer residency in the GGSN remains low.

The protocol implementation provides a message queue based interface to higher layers: messages are placed in a queue for transmission and retrieved from a queue after receipt. Within the protocol stack, messages are serialised and split into segments before transmission, and segments reassembled into messages on receipt.

6.3 Caching

GPRSWeb implements an extended caching scheme intended to optimise the hit rate of the client cache, and thus minimise page download time and reduce bandwidth requirements. In terms of the freshness of pages actually returned for the user, the cache is no more aggressive than allowed by the normal HTTP algorithm, unless the GPRS link is currently down in which case the client proxy can be configured to return potentially stale data to allow something to be displayed.

The GPRSWeb extended caching protocol indexes objects by their SHA-1 fingerprint (content hash). A separate table is maintained that maps URLs to the respective Content Hash Key (CHK). This enables multiple URLs pointing to identical documents to be stored just once in the cache. In some dynamically generated web sites such identical mappings are commonplace, resulting in significantly improved hit rates.

The client cache is intended to replace the browser's persistent cache. During the course of proxy installation, the browser's persistent cache is disabled and flushed. The browser's in-memory cache is left enabled for performance reasons.

Each **Cache Entry** contains a document body and the time it was last used, stored in a file named by the document CHK. The **Cache Index** maintains an in-memory list of cache entry metadata. It is initialised with data read from the cache entries on disk, and is updated alongside the on-

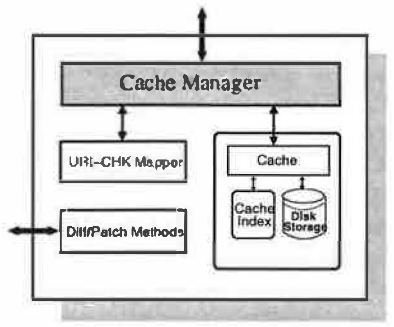


Figure 8: Cache Operation Overview

disk cache. The **URL Mapper** maintains mappings between URLs and the CHKs representing their bodies. Associated with each entry are the original HTTP Response headers, used to construct a Response when servicing a request from the cache. This allows multiple responses to share the same body, but with different headers.

When a URL mapping expires (as indicated by the conventional HTTP caching mechanism), it must be refreshed before being used again. The client proxy asks the server proxy to do this on its behalf.

The server proxy will check its own cache to see whether a more recent mapping exists. This can occur if another of its clients has accessed the same object. If not, it will have to contact the server to either check the modification time or fetch the object. Often, as a result of the pessimistic caching directives returned by sites with dynamic content, the object turns out to be identical to the previous version. The server proxy indicates this to the client by simply retransmitting the URL to CHK mapping, along with any caching directive returned by the server. In fact, thanks to the 'parse and push' mechanism described later the client often does not even need to request that a mapping be refreshed because the server proxy will have pro-actively sent a message containing the refreshed mappings.

The server proxy tracks the state of each client proxy's cache by modelling the replacement policy of the client, which is 'least recently used' eviction. The contents of its own cache is a superset of its clients'. If synchronisation is lost, for example, if the client proxy is directed to connect to a new server, the client uses low priority messages to update the server on its cache status.

6.4 Delta-encoding and Compression

The GPRSWeb proxies attempt to ensure that all data travelling over the GPRS link is in a compressed form, to reduce transfer size and improve response time. Unless the data is already in a compressed form (for example JPEG images or Zip archives), the gzip compression algorithm is employed.

Where the data being sent is an updated version of a previous object (same URL, different CHKs) a 'delta encoding' [15] algorithm is tried. Delta encoding sends differences between new and old versions of a document. The strategy is often very successful as many updated documents are very similar to their predecessor, particularly for dynamically generated content. A classic example is news site front pages that contain a string indicating the time of day.

Since the server proxy tracks the contents of the client cache is able to use the VCDiff [31, 32] algorithm to produce the *deltas* from a document it knows the client has. The deltas are gzip compressed and sent to the client if the resulting data is smaller than sending a compressed version of the new object.

Similar compression mechanisms are used for HTTP headers, both requests and responses. A separate string table is used for HTTP headers to avoid useful strings being evicted during the transfer of object data. This approach is very successful, since HTTP headers produced by modern browsers are rather verbose, and the variation between requests of the same type is small.

6.5 Parse-and-Push Operation

As discussed earlier, most web pages contain a number of images and other support objects (frames, style sheets etc.) that make up the page structure. These are eventually requested by the browser after parsing the HTML document.

The parse-and-push mechanism in the server proxy attempts to speculatively push toward the client objects and URL to CHK mappings that it knows are going to miss in the client cache. These are sent with lower priority than responses to requests explicitly made by the client. Responses are promoted if an explicit request is received for them. The main benefit from the parse-and-push mechanism is to keep the link utilized during delays due to the data dependencies between object references. For pages with complex layouts these can be quite significant – as well as the network RTT delay, it can take the browser some time to process the returned HTML.

The parsing performed by the server proxy is currently crude, but fast. Documents of type text/html are parsed using a regular expression to extract references to support objects. Duplicates are removed, and relative URLs combined with the document base to produce a list of candidate URLs. The server proxy may miss some object references (these will be simply be requested later by the client), and may in fact construct some invalid URLs. These will typically result in "URL not found" error codes when the proxy attempts to fetch them from the server, and will not be pushed to the client.

6.6 Image Transcoding

All the optimization techniques described up until now are "loss-less": they do not change the appearance of the page returned to the user. We have also experimented with a simple image transcoding scheme to shrink the size and quality of JPEG images sent over the wireless link.

Other groups have investigated the utility of transcoding, and mechanisms for its implementation far more thoroughly than us e.g. [11]. We included this functionality in the proxy as a placeholder for future work. In the experiments described here, the client proxy returns the image to its original width and height before passing it to the browser (though it is obviously somewhat degraded). A more complete implementation would degrade the image at the server proxy under the control of browser, using hints contained in the content.

7. GPRSWEB SYSTEM PERFORMANCE

7.1 Implementation

We have designed and implemented the GPRSWeb proxy system system over Windows XP/Windows 2000. The GPRSWeb server and client proxy was written in C# (Csharp) over Microsoft's .NET framework. C# is a new type-safe and garbage collected language with a powerful function library (especially for *networking*) to speed up project development. C# is also supported on WinCE based devices such as PDAs and smart-phones. We intend to port the client proxy code to such devices in the future. The client and server proxies share much of the source code for the GPRSWeb protocol stack and cache interface functionality.

7.2 Experimental Test Setup

Our experimental set-up for evaluating the GPRSWeb proxy system is shown in figure 9. The mobile terminal (laptop) was connected to the GPRS network via a Motorola T260 GPRS phone (supporting 3 downlink and 1 uplink channels). The GPRS network was provided by Vodafone. The GPRSWeb client software was installed on the mobile terminal, a laptop running Windows XP.

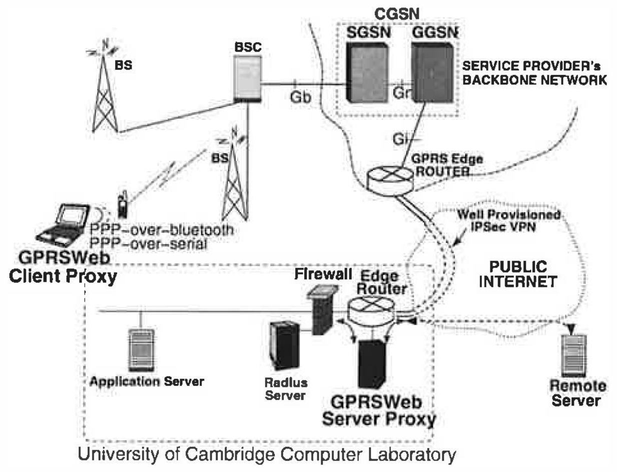


Figure 9: Experimental Test Bed Set-up

Since we were unable to install equipment to run the GPRSWeb proxy server next to the CGSN, we made use of a well provisioned IPSec VPN to 'back haul' GPRS traffic to the lab. The proxy server ran on a Windows 2000 server located near to the tunnel endpoint.

7.3 Experimental Results

In this section, we present an evaluation of how well the GPRSWeb proxy system improves WWW performance over GPRS. Specifically, we attempt to quantify the overall performance benefit relative to an unassisted browser. These experiments were performed using a single, stationary, mobile client to minimise variation in GPRS link performance.

We used the Mozilla 1.0 browser for recording these results. This was chosen due to the availability of source code, enabling us to instrument the browser to log web page download times. In non-persistent mode, Mozilla employs up to 8 parallel connections. In persistent connection mode, it uses up to 6 simultaneous connections. GPRS performance with other web browsers (Internet Explorer 6, Netscape 6) seems very similar to that of Mozilla, though we have not evaluated them as thoroughly.

7.3.1 Test Web Sites

To evaluate the performance benefits of our scheme, we performed experimental downloads of two synthetically arranged web pages offering static content, and also snapshot of the front page of two popular `news` web-sites and an `e-commerce` web site.

We arranged for these test pages to be hosted on a local server, eliminating the performance vagaries of public networks and servers. Furthermore we were able to control when content on these local pages was updated.

To create the synthetic pages, we adhered to the approach used in [19], composing a number of objects from other websites into a single page according to object type and file size distributions observed in HTTP proxy log traces. We consider two types of web site `STATIC-I` and `STATIC-II` (see table 1 and 2). The first is a relatively simple page consisting of a base HTML document with a few jpeg/gif images. The second one represents a more complex page comprising 46 objects.

For real-test web pages, we created mock-up of two popular news web sites, `www.cnn.com` and `www.bbc.co.uk`, and a third e-commerce web-site `www.amazon.com` based on a snapshot of their front page. We term them here as `1CNN`, `1BBC`, and `1AMAZON` respectively. These web-pages consisted of over 50 embedded objects including cascading style sheets (.css), active server pages (.asp) and java scripts (.jss).

Resource Type	Size/ Size Range	Total Number of files/objects	% w.r.t total content
index.html	17KB	1	52%
jpegs	2KB-4KB	3	29%
gifs	200B-5KB	5	19%

Table 1: `STATIC-I` web page composition

Resource Type	Size/ Size Range	Total Number of files/objects	% w.r.t total content
index.html	40K	1	25%
jpegs	2KB-5KB	8	17%
gifs	200B-2KB	24	17%
gifs	2KB-10KB	12	34%
gifs	>10KB	1	7%

Table 2: `STATIC-II` web page composition

7.4 Performance Evaluation

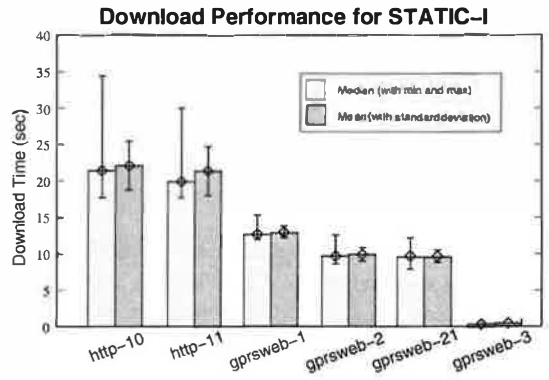
We present results comparing the performance downloading the test pages using the unassisted browser *vs.* using the GPRSWeb proxy. We used the default setting of the GPRSWeb proxy, that employs the full set of loss-less optimization techniques: *the enhanced transport protocol*, *data compression*, *delta encoding*, *extended caching* and *parse-and-push*.

Additionally, a separate experiment was performed with image transcoding also enabled. The transcoder module degraded only JPEG images, and only by a small amount, resulting in a typical image transfer size reduction of about 10%.

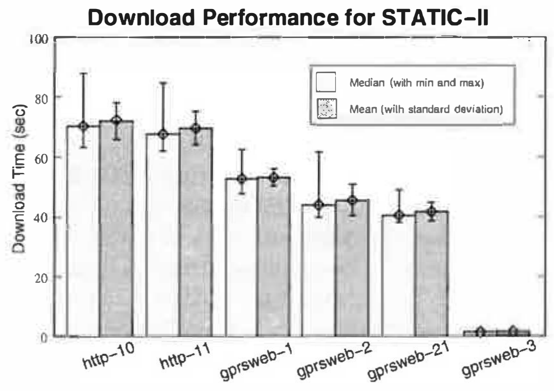
Except where stated, we flushed all client caches before each download test. We report results with both the server-side cache 'hot' and 'cold'.

We evaluated the following scenarios:

- `http-10`:- We measured download times using Mozilla over GPRS in non-persistent (HTTP/1.0) mode operating directly with the server.
- `http-11`:- These measurements were taken with Mozilla operating directly with the server in persistent connection (HTTP/1.1) mode.
- `gprsweb-1`:- These measurements are taken with the browser using the GPRSWeb proxy, but with cold client and server-side caches. Image transcoding was disabled.
- `gprsweb-2`:- Similar to `gprsweb-1`, but with a hot server-side proxy cache from which all objects are able to be served.
- `gprsweb-21`:- The scenario is similar to the `gprsweb-2`, but with image transcoding enabled.
- `gprsweb-3`:- Represents the best case scenario – a hit at the local client cache.



(a) STATIC-I



(b) STATIC-II

Figure 10: Mean download times with (top-down) (a) `STATIC-I`, (b) `STATIC-II` measured from 30 successful runs

For each of the scenario discussed above, we recorded download times from 30 successful runs and plot the mean (average) value of the download times and corresponding standard deviation. For additional clarity, we also plot the median along with minimum and maximum value of the download time from each of the data set.

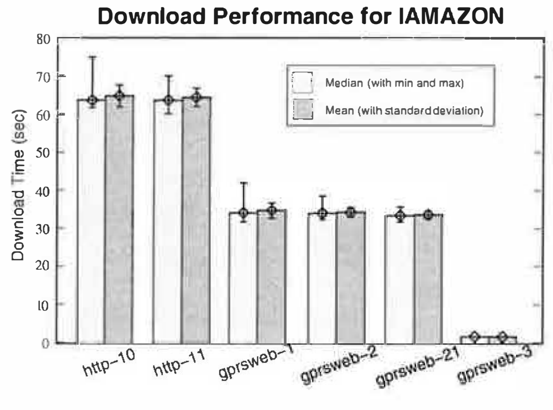
Figure 10(a) shows the mean download times for `STATIC-1`. We observe that use of HTTP/1.1 offers only meagre benefit (3-5%) over HTTP/1.0. However, we see a major improvement in download times using GPRSWeb protocol. Even with a cold server-side cache (`gprsweb-1`) the use of GPRSWeb protocol leads to 40%-45% reduction in mean page download times over GPRS. In the `gprsweb-2` case where the server-side cache is warm we record a performance improvement of 55%-60%.

Finally, the test where all objects hit in the local client proxy cache obviously gives the best performance. Page rendering time is broadly similar to that of a browser accessing content from its own local persistent cache.

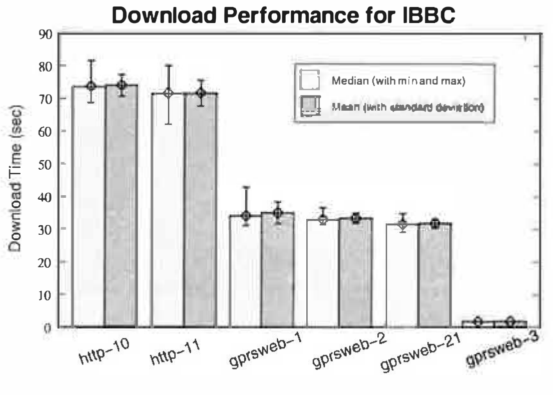
For the cold server-side cache case, the results from `STATIC-1` are reflected for other test pages – `1AMAZON` shows a mean reduction of about 45-50% while `1BCC` gives a benefit of over 50-55% (see figure 11 (a) and (b)). On the other hand, the benefits provided by GPRSWeb were not quite so substantial for `STATIC-II` and `1CNN`. In the cold server-side cache case GPRSWeb offers an improvement of 26% and 29% in page download times for `STATIC-II` and `1CNN` web pages respectively. With a warm cache, however, we see an encouraging reduction of 35-40% in mean page download time.

In these tests, our simple image transcoding optimization shows little benefit, and in fact the image processing delay actually makes matters worse in the `1CNN` case. In the `STATIC-II` page where there are several JPEG images some small advantage is shown. This is similar even for `1AMAZON` and `1BBC`. To properly evaluate the potential of image transcoding we should probably degrade GIF and PNG images as well as JPEGs, and use a more aggressive quality reduction settings.

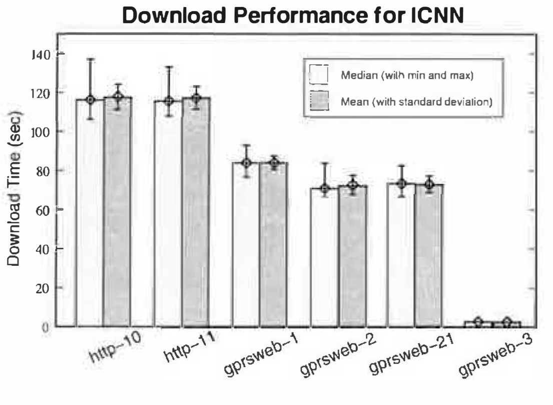
Based on these results, we believe that the GPRSWeb proxy is capable of substantial reductions in web page download time. However, the experiments presented are yet to exercise the proxy's CHK-based caching or delta-encoding of



(a) lAMAZON



(b) lBBC



(c) lCNN

Figure 11: Mean download times of some commercially available web-sites with (top-down) (a) lAMAZON (e-commerce), (b) lBBC (news), and (c) lCNN (news) measured from over 30 successful runs

object versions. This can only be done using a web site with changing, dynamically generated content. Experience using the proxy for real-life web browsing over GPRS suggest that these optimizations come into play quite frequently, and result in substantial performance wins when they do so.

We are in the process of building an experimental setup that enables recorded traces of user browsing behaviour to be accurately replayed, with the server reflecting the different versions of the content that were delivered on different occasions. This should enable an accurate evaluation of the real-world practical benefit of the extended caching and delta-encoding schemes.

8. RELATED WORK

A variety of previous research has examined techniques that can be used to optimise web browsing over high-latency links, but none has looked at GPRS links specifically. Related work can be broken into three categories: solutions that look at improving wireless web performance; transport enhancements that elevate TCP performance over wireless links, and finally other optimization schemes (e.g. smart caching, prefetching etc.) aimed at reducing data transferred and round trips made over the link.

8.1 Wireless Web Solutions

A number of significant research studies have investigated web performance, in general [18, 19, 15], and more specifically, wireless web performance [22, 12, 20, 21, 35].

Mogul *et al.* [18] discuss HTTP throughput and latency problems and show that each document and inlined image requires a minimum of ($2 \times$ RTT) for transfer. They propose the use of persistent connections and pipelining to improve overall web performance. Nielsen *et al.* [19] clearly demonstrate benefits of pipelined implementations, while Mogul *et al.* [15] argue that use of delta encoding and compression between client browser and a proxy can 'remarkably' improve web performance.

Liljeberg *et al.* [22] developed Mowgli Communications Architecture that uses a pair of proxies and employs its own protocol over the link. Mowgli is similar to GPRSWeb but focuses on GSM networks, and gives more weight to disconnected operation than we do for GPRS. It uses its proprietary protocol called Mowgli HTTP (MHTTP) that improves limitations of TCP and HTTP over GSM. The MHTTP protocol optimizes use of bandwidth using binary header encoding, data compression of text and images, and image transcoding.

WebExpress [12] from IBM focuses on improving web browsing performance over wireless links through caching, differencing, protocol and header reduction mechanisms. Unlike WebExpress, GPRSWeb focusses on the "always-on" GPRS link and specifically on web performance. This is somewhat different from WebExpress, which aims at improving web *form* performance on GSM links. WebExpress was designed to support applications characterized by repetitive and predictable responses, where only some certain information change in a given page.

Fleming *et al.* in [20] use a similar scheme to that used in

Wireless Web Solutions	Transport Protocol	Cach-ing	Client prefetch	Parse-and-Push	Fast start and DNS Migration	Compre-ssion	Filtering/ Transcoding	Delta-enconding and CHK
Mowgli [22]	Custom	√	×	×	√	√	√	×
WebExpress [12]	TCP	√	×	×	×	×	protocol/header reduction only	√ (no CHK)
MHSP [20]	TCP	√	√	×	×	×	√	×
FirstHop [35]	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.
GPRSWeb	**Custom**	√	√	√	√	√	√	√

Table 3: Feature Summary of Wireless Web Solutions [N.A. - Not Available]

Mowgli, but also include prefetching schemes in their wireless world wide web proxy server and protocol using their new Multiple Hypertext Stream Protocol (MHSP). H. Balakrishnan *et al.* [21] use explicit loss notification (ELN) to improve web performance using a scheme by which senders are informed about the cause of loss event – network congestion or bit error. Thus ELN decouples sender retransmissions from congestion control.

Commercial products that improve upon the current GPRS performance are also available. A GPRS Accelerator under development by Firsthop [35] claims faster data transfers over GPRS. Their approach is to reduce data exchange and optimize protocols appropriately over the wireless link. However, it is unclear how this data reduction is achieved - using delta encoding and compression [15] or other sophisticated data (and header) compression techniques or by simply filtering/transcoding at the proxy. Nevertheless, their approach seems to necessitate a client-side software update. This obviously points us to believe that it might be making use of caching and/or delta compression/data transcoding.

8.2 Transport Layer Enhancements

In this section, we briefly review transport layer enhancements for wireless networks. Again, this can be factored into TCP, and non-TCP based solutions, which are generally UDP based. We briefly discuss some of these solutions to improve transport performance over wireless.

Snoop [23] is a TCP aware link-layer scheme that sniffs packets in the base station and buffers them. If duplicate acknowledgements are detected, incoming packets from the mobile host are retransmitted (if present) from the local cache. On the wired side, dupacks are suppressed from the sender, thus avoiding unnecessary fast retransmissions and the consequent invocation of congestion control mechanisms. However, Snoop was designed for wireless LANs rather than for 'long-thin' wide-area wireless links such as GPRS. As such, it does not address the problems of excess queueing at the base stations or proxies.

The 'split TCP' approach (such as the one used in [8]) is quite popular since it allows wireless losses to be completely shielded from the wired ones. I-TCP [24] uses a similar scheme for TCP over the wireless link, albeit with only some modifications. However, as TCP in I-TCP is not tuned for wireless links, it often leads to timeouts eventually causing stalls on the wired side and poor link utilization.

Then there are other schemes for improving TCP performance that make use of some sort of early warning signals. For example, Freeze-TCP [25] uses a proactive scheme in which a mobile host detects signal degradation and sends a Zero Window Probe (ZWP). The warning period – the time before which actual degradation occurs should be sufficient for the ZWP to reach the sender so that it can freeze its window. A drawback here is the reliability of this calculation and Freeze-TCP's inability to deal with sudden random losses. Furthermore, such schemes necessitate end-system changes.

Non-TCP based solutions typically optimize connection set-up/teardown and control overhead associated with TCP. One such example is Wireless Application Protocol (WAP) [28], in which a WAP gateway splits the transport with a new protocol for use over the wireless link. It uses two different protocols – Wireless Transport Protocol (WTP) and Wireless Datagram Protocol (WDP) over wireless, while using standard TCP over wired networks. WDP offers a general datagram service to the upper layer protocol to assist in communicating transparently with bearer services. Unlike WDP, WTP improves reliability of datagram services and relieves the upper layer from re-transmissions. It uses a message based transaction oriented protocol that assist in activities like web browsing.

8.3 Other Optimizations

A number of different studies have investigated cache performance. A hit at the client cache eliminates the need to traverse the high RTT wide-area wireless link. Some caching approaches are detailed in [26]. Cache Digests in [27] are a quick way of sending details about cache contents between caches – useful for synchronising caches over high latency links.

Web prefetching over networks, deterministic or predictive, is generally accepted to be useful. However, it is questionable if client-side predictive pre-fetching schemes over GPRS would be advantageous. The assumption made in earlier studies (e.g. Fleming *et. al.* [20]) was that since the link is idle anyway, why not use it for downloads, even if the prefetched page is oftentimes not useful. However, the links where this has been evaluated have employed *time-based* charging rather than the *volume* based charging typically used by GPRS operators. Since GPRS bandwidth is typically at least an order of magnitude more expensive than fixed-Internet bandwidth, the tradeoffs may be rather different.

The *parse-and-push* mechanism employed by GPRSWeb has some similarities to web prefetching, except the set of object pushed to the client are deterministic (and known to be of use to the client), and confined to support resources for the current page.

9. ISSUES AND DISCUSSION

In this section, we discuss some key issues relating to web optimization over GPRS.

9.1 Adaptation in Web Browsers

An issue crucial to WWW performance is how web browsers can be made to adapt to underlying network heterogeneity. As TCP connections continue to span a number of disparate links – wired as well as wireless – browser performance will increasingly depend upon how they attune their performance to adapt to the underlying network.

There are two approaches to optimizing browsing experience over heterogenous networks – browser adaptation, or a local proxy-based solution. With more penetration of WLANs and wide-area GPRS/3G, we feel that browsers of the future will have to be designed such that they can adapt to the underlying network. This can be a smart adaptation based on the type of the underlying network. Lower layers can indicate a change in the network, which in turn can assist browsers to quickly adapt to the new network.

9.2 GPRSWeb Server Proxy Location

In this section, we discuss possible locations for deployment of proxy servers within a GPRS network. A number of possibilities exist (see, figure 12): (a) close to the gateway router of the cellular provider where traffic density is likely high (in figure *Proxy*(1)), (b) near to the wired-wireless boundary (i.e. GPRS CGSN node or close) where traffic from a number of mobile hosts is aggregated (see *Proxy*(2)), (c) close to GPRS SGSN node where traffic from a number of base stations can be handled (shown as *Proxy*(3)), or (d) in the vicinity of the base station that handles traffic from a single cell (see, *Proxy*(4)).

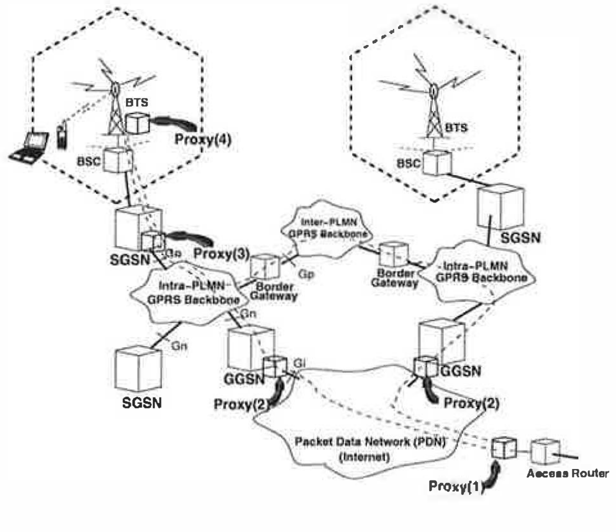


Figure 12: Possible Server Proxy Locations

In general, by placing proxies further inside a cellular network, one can get much better access to information about current radio link conditions than proxies located outside the network. Proxies located close to the base station can be informed of dynamically varying channel conditions, and could take action based on this. If fine-grained channel monitoring is not required, as in the case of GPRSWeb server proxy, it can be safely placed at, or close to the GPRS CGSN node, able to serve a large number of mobile clients from a single location.

9.3 Security Issues

GPRSWeb currently does not offer any additional encryption or authentication mechanisms. It relies on the security of the underlying GPRS network. A drawback using custom protocol over GPRS in GPRSWeb is that it can not provide end-to-end security (e.g. https). Designing a proxy-based solution that allows end-to-end security is still an open topic for research. However, protocols such as the Wireless Transport Layer Security (WTLS) protocol could be used to provide privacy, data integrity, and authentication for the link between the proxies. WTLS closely resembles Secure Socket Layer (SSL)/Transport Layer Security (TLS) protocol, yet is optimized for use over low bandwidth wireless links and for resource constrained mobile devices. WTLS supports datagram oriented protocols, so porting GPRSWeb would be straight forward.

Furthermore, the current GPRSWeb server proxy would be quite vulnerable to denial of service attacks, since it provides resource intensive services. The danger could be limited to some extent by ensuring that only clients from the wireless network can connect to the proxy, thus limiting the load an individual client can generate in an attempt to starve others.

10. CONCLUSION AND FUTURE WORK

In this paper, we explored the causes of TCP and HTTP under-performance in the GPRS environment. Due to GPRS' high latency, the need for latency mitigation was highlighted, leading to the design of the GPRSWeb proxy system, a pair of co-operating proxies positioned either side of the wireless link.

We have described our implementation and results of an initial performance evaluation of the prototype system. We have shown that the collective suite of optimization techniques implemented by GPRSWeb can lead to substantial reductions in mean page download times.

Currently, we have used the proxy system with only a limited set of concurrent clients. It would be interesting to perform tests (part of our ongoing work) to identify how well the server proxy scales to a wider client base. We are currently distributing GPRSWeb client software to a number of campus GPRS users. We are recording full tcpdump traces of all GPRS traffic generated by our user community, which will assist in evaluating system scalability and resulting user experience from using the proxy system.

In other ongoing work, we plan a thorough evaluation of the practical benefits of GPRSWeb's extended CHK-based caching and delta-encoding schemes. We intend to use our traces of user browsing activity and the corresponding server responses to accurately replay user activity, enabling us to precisely evaluate the performance gain these techniques can offer in the presence of real dynamically changing web content.

11. ACKNOWLEDGEMENTS

We would like to thank Vodafone Group R&D, Sun Microsystems Inc. and BenchMark Capital for supporting this work. Thanks also go to Tim Harris for his comments on this paper, and to Cristina-Ana-Maria Hristea for shepherding the submission of the final version of this paper. We also thank the anonymous reviewers for many constructive inputs. The complete source code for GPRSWeb including the test web-sites used in this paper is available:

`http://www.cl.cam.ac.uk/users/rc277/soft.html`

12. REFERENCES

[1] G. Brasche and B. Walke, "Concepts, Services and Protocols of the New GSM Phase 2+ General Packet Radio Service", *IEEE Communications Magazine*, August 1997.

[2] R. Chakravorty and I. Pratt, "Performance Issues with General Packet Radio Service (GPRS)", Journal of Communications and Networks (JCN), pages 266-281, Vol. 4, No. 2, December 2002 (ISSN 1229-2370). In the Special Issue of *Evolving from 3G deployment to 4G definition* `http://www.cl.cam.ac.uk/users/rc277/gprs.html`

[3] R. Chakravorty, J. Cartwright and I. Pratt, "Practical Experience With TCP over GPRS", In *Proceedings of IEEE GLOBECOM 2002*, November 17-21, Taipei, Taiwan `http://www.cl.cam.ac.uk/users/rc277/gprs.html`

[4] M. Meyer, "TCP Performance over GPRS", In Proceedings of IEEE WCNC, pages 1248-1252, 1999

[5] C. Bettssetter, H. Vogel, J. Eberspacher, "GSM Phase 2+ General Packet Radio Service GPRS: Architecture, Protocols, and Air Interface", *IEEE Communication surveys* Third Quater 1999, Vol.2 No.3.

[6] R. Ludwig et al., "Multi-Layer Tracing of TCP over a Reliable Wireless Link", In Proceedings of ACM SIGMETRICS 1999.

[7] R. Chakravorty and I. Pratt, "WWW Performance over GPRS", In *Proceedings of IEEE International conference in Mobile and Wireless Communications Networks (IEEE MWCN 2002)*, September 9-11, Stockholm, Sweden `http://www.cl.cam.ac.uk/users/rc277/gprs.html`

[8] R. Chakravorty, S. Katti, J. Crowcroft and I. Pratt, "Flow Aggregation for Enhanced TCP over Wide-Area Wireless", In Proceedings of INFOCOM 2003, San Francisco (to appear) `http://www.cl.cam.ac.uk/users/rc277/gprs.html`

[9] H. Balakrishnan et al., "A Comparison of Mechanisms for Improving TCP Performance over Wireless Links", IEEE/ACM Trans. on Networking, Vol. 5, No.6, Dec. 1997.

[10] Z. Wang and P. Cao, "Persistent Connection Behaviour of Popular Browsers", http://www.cs.wisc.edu/cao/papers/persistent-connection.html

[11] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, "Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspective", in IEEE Personal Communications, Vol. 5, No. 4, pages 10-19, August 1998

[12] B. C. Housel and D. B. Lindquist, "WebExpress: A System for Optimizing Web Browsing in a Wireless Environment", In *Proceedings of ACM MOBICOM*, November 1996.

[13] M. Liljeberg et al., "Optimizing World-Wide Web for Weakly-Connected Mobile Workstations: An Indirect Approach", In *Proceedings of 2nd International Workshop on Services in Distributed and Networked Environments (SDNE)*, pages 132–129, Whistler, Canada, 1995

[14] J. C. Mogul, "Support for out-of-order responses in HTTP", *Internet Draft*, Network Working Group, 6 April 2001.

[15] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential benefits of delta encoding and data compression for HTTP", In *Proceedings of ACM SIGCOMM*, pages 181-194. Cannes, France, September 1997.

[16] R. Stewart et al., "T/TCP – TCP Extensions for Transactions", *Request for Comments (RFC) – 2960*, October 2000.

[17] R. Braden et al., "Stream Control Transmission Protocol", *Request for Comments (RFC) – 1644*, July 1994.

[18] V. N. Padmanabhan and J. C. Mogul, "Improving HTTP Latency", *Computer Networks and ISDN Systems*, vol. 28, pages 25-35, 1995

[19] H. Nielsen, J. Gettys, A Baird-Smith, E. Prud'hommeaux, H. Lie and C. Lilley, "Network Performance Effects of HTTP/1.1 CSS1, and PNG", In *Procedings of ACM SIGCOMM*, Cannes, France, September 1997

[20] T. B. Fleming, S. F. Midkiff, and N. J. Davis, "Improving the Performance of the World Wide Web over Wireless Networks", In *Proceedings of IEEE GLOBECOM*, 1997

[21] H. Balakrishnan and R. H. Katz, "Explicit Loss Notification and Wireless Web Performance" In *Proceedings IEEE Globecom Internet Mini-Conference*, Sydney, Australia, November 1998.

[22] M. Liljeberg, H. Helin, M. Kojo, and K. Raatikainen, "MOWGLI WWW Software: Improved Usability of WWW in Mobile WAN Environments", *IEEE Global Internet*, November 1996

[23] H. Balakrishnan, R. Katz and S. Seshan, "Improving TCP/IP performance over Wireless Networks", In *Proceedings of ACM MOBICOM*, November 1995

[24] A. Bakre and B. R. Badrinath, "I-TCP: Indirect TCP for mobile hosts", In *Proceedings of the 15th IEEE ICDCS*, pages 136-143, Vancouver, BC, May 1995.

[25] T. Go , J. Moronski, D. S. Phatak, and V. Gupta, "Freeze-TCP: A true end-to-end enhancement mechanism for mobile environments," In *Proceedings of IEEE INFOCOM 2000*, Israel.

[26] G. Barish and K. Obraczka, "World Wide Web Caching: Trends and Techniques", *IEEE Communications Magazine*, Internet Technology Services, May 2000.

[27] L.Fan, P Cao, J. Almeida and A. Z. Broder, "Summary Cache: A scalable wide-area web cache sharing protocol", Technical Report 1361, Department of Computer Science, University of Wisconsin – Madison, Feb 1998.

[28] `http://www.wapforum.org`

[29] NIST, FIPS PUB 180-1: "Secure Hash Standard", `http://www.itl.nist.gov/fipspubs/fip180-1.html`, April 1995

[30] NZipLib, The .NET Zip Library `http://www.icsharpcode.net/OpenSource/NZipLib/`

[31] Internet Draft, "The VCDiff Generic Differencing and Compression Data Format", http://www.ietf.org/internet-drafts/draft-korn-vcdiff-06.txt, November 2001

[32] JLibDiff, Java Diff Library `http://sourceforge.net/projects/jlibdiff/`

[33] J. Cartwright, "GPRS Link Characterization", `http://www.cl.cam.ac.uk/users/rc277/linkchar.html`

[34] "An Introduction to the Vodafone GPRS Environment and Supported Services", Issue 1.1/1200, December 2000, Vodafone Ltd., 2000.

[35] Firsthop GPRS Accelerator, `http://www.firsthop.com/`

[36] `tcpdump(http://www.tcpdump.org)`, `tcptrace(http://www.tcptrace.org)`, `ttcp+(http://www.cl.cam.ac.uk/Research/SRG/netos/netx/)`

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

For a complete listing of member benefits, see http://www.usenix.org/membership/bens.html.

Want more information about USENIX? See http://www.usenix.org/ or contact:
USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

USENIX & SAGE Thank Their Supporting Members

USENIX Supporting Members

❖ Ajava Systems, Inc. ❖ Aptitune Corporation ❖ Atos Origin B.V. ❖
❖ Computer Measurement Group ❖ Freshwater Software ❖ Interhack Corporation ❖
❖ The Measurement Factory ❖ Microsoft Research ❖ Sun Microsystems, Inc. ❖
❖ UUNET Technologies, Inc. ❖ Veritas Software ❖ Ximian, Inc. ❖

SAGE Supporting Members

❖ Certainty Solutions ❖ Freshwater Software ❖ Microsoft Research ❖ Ripe NCC ❖

ACM SIGMOBILE

ACM SIGMOBILE is the Association for Computing Machinery's Special Interest Group on Mobility of Systems, Users, Data, and Computing. Founded in 1947, ACM is the world's first educational and scientific computing society. Today, ACM serves a membership of more than 80,000 computing professionals in more than 100 countries in all areas of industry, academia, and government.

SIGMOBILE is the primary international organization dedicated to addressing the latest developments in all areas of mobile computing and wireless and mobile networking. SIGMOBILE has members from around the world, from academic organizations, industry research and development, government, and other interested individuals. Members of SIGMOBILE are active in the development of new technologies and techniques for mobile and wireless computing and communications.

As a member of SIGMOBILE, you will receiver our quarterly journal, *Mobile Computing and Communications Review* (MC^{2}R). You will also be able to receive announcements via our moderated members-only email distribution list, keeping you informed of the latest happenings in our field, including new opportunities to share your research and to meet with friends and colleagues at conferences and other events. SIGMOBILE members also are eligible for the lowest generally available registration fee for any event sponsored or co-sponsored by SIGMOBILE, and for the many events sponsored by other organizations offered in-cooperation with SIGMOBILE.

For more information about ACM SIGMOBILE, visit us on the web at *http://www.sigmobile.org/*. You may also contact the ACM Membership Services Department by email at *acmhelp@acm.org* or by telephone at 1-800-342-6626 (U.S. and Canada) or +1-212-626-0500 (outside U.S.).

ISBN 1-931971-09-9